(Un)Safe Browsing

Thomas Gerbet, Amrit Kumar, Cédric Lauradoux

(Un)Safe Browsing

Thomas Gerbet, Amrit Kumar, Cédric Lauradoux

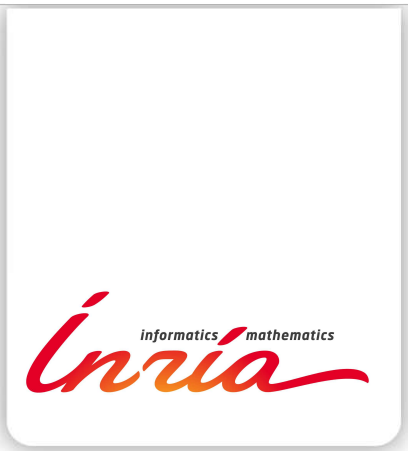
RESEARCH REPORT

N° 8594

September 2014

Project-Team Privatics

ISSN 0249-6399 ISRN INRIA/RR--8594--FR+ENG



(Un)Safe Browsing

Thomas Gerbet*, Amrit Kumar, Cédric Lauradoux

Project-Team Privatics

Research Report n° 8594 — September 2014 — 13 pages

Abstract: Users often accidentally or inadvertently click malicious phishing or malware website links, and in doing so they sacrifice secret information and sometimes even fully compromise their devices. These URLs are intelligently scripted to remain inconspicuous over the Internet. In light of the ever increasing number of such URLs, new ingenious strategies have been invented to detect them and inform the end user when he is tempted to access such a link. The *Safe Browsing* technique provides an exemplary service to identify unsafe websites and notify users and webmasters allowing them to protect themselves from harm. In this work, we show how to turn Google Safe Browsing services against itself and its users. We propose several *Distributed Denial-of-Service* attacks that simultaneously affect both the Google Safe Browsing server and the end user. Our attacks leverage on the *false positive probability* of the data structures used for malicious URL detection. This probability exists because a trade-off was made between Google's server load and client's memory consumption. Our attack is based on the forgery of malicious URLs to increase the false positive probability. Finally we show how Bloom filter combined with universal hash functions and prefix lengthening can fix the problem.

Key-words: Hash functions, denial-of-service, pollution

* Université Joseph Fourrier, Grenoble

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un)Safe Browsing

Résumé : De nos jours, les utilisateurs cliquent accidentellement ou par inadvertance sur des sites malveillants de phishing ou contenant des malwares. Ce faisant, ils sacrifient des informations secrètes et parfois même compromettre pleinement leur objets. Les URLs de ces sites sont intelligemment scénarisés pour passer inaperçu et tromper leur monde quand Ã leur contenu. Compte tenu du nombre sans cesse croissant de ces URLs, de nouvelles stratégies ingénieuses ont été inventés pour les détecter afin d'informer l'utilisateur final lorsqu'il est tenté d'accéder Ã un tel lien. Google Safe Browsing est un mécanisme de navigation permettant d'identifier les sites Web dangereux et avertir les utilisateurs et les administrateurs en leur permettant de se protéger. Dans ce travail, nous montrons comment retourner Google Safe Browsing contre lui-même et de ses utilisateurs. Nous proposons plusieurs attaques de déni de service distribué qui affectent simultanément le serveur de Google Safe Browsing et l'utilisateur final. Notre attaque s'appuie sur l'existence d'une probabilité de faux positifs dans la structure de données utilisée pour la détection d'URL malveillantes. Cette probabilité existe parce qu'un compromis a été choisi par Google entre la charge du serveur et la consommation de mémoire du client. Notre attaque est basée sur la génération de sites malveillants ayant des URLs particulières afin d'augmenter la probabilité de faux positifs. Enfin, nous montrons comment les filtres de Bloom combinés avec des fonctions de hachage universelles et des préfixes rallongés peuvent résoudre le problème.

Mots-clés : Fonction de hachage, déni de service, pollution

1 Introduction

Communications purporting to be from popular social web sites, auction sites, banks, online payment processors or IT administrators are rampantly used to lure unsuspecting public. These attempts popularly known as phishing, aim to acquire sensitive information such as usernames, passwords, and credit card details by masquerading as a trustworthy entity. In general, these attacks rely on seemingly authentic URLs to entice a user to transmit sensitive data to the back-end malicious entities or to install harmful piece of code *aka* malware on their device.

These malicious URLs are often hard to be visually detected, whence, the *raison d'être* for more reliable, automated and robust way of detecting them. *Safe Browsing* is such a service provided by Google that enables Chrome to check a URL against suspected phishing or malware pages' database. The browser can then warn the user about the potential danger of visiting the page. Later, other browsers including Firefox, Opera and Safari have included Google Safe Browsing as a feature. Google claims more than a billion users of Google Safe Browsing until date [**?**]. Despite its success, it was however criticized in 2013, as it used a cookie which could be exploited for tracking. An in-depth assessment of the security and the privacy implications of the mechanism is required and constitutes the motivation of our work.

In this paper, we describe several attacks against two versions of Google Safe Browsing. For all our attacks, we explain how an adversary carefully chooses the URL of his malicious web page to increase the load of Google's servers and to make browsers consume more bandwidth and more CPU resources. Conceptually, our goal is to mount *distributed denial-of-service* (DDoS). DDoS attacks pose an immense threat to the Internet. The field of DDoS is rapidly becoming more and more complex, and has reached the point where it is difficult to see the forest for the trees. An extensive study of network-based DDoS attacks and defense mechanisms is provided in [**?**, **?**]. To this end, we provide new DDoS attacks that do not rely on any network resources, such a *botnets*, *secondary victims* or *agents*. Yet, the hosts at both ends of the communication network, i.e. the safe browsing server and its clients are simultaneously attacked. Classical detection methods [**?**, **?**, **?**, **?**, **?**] are useless because all users in our attack are acting honestly. Our DDoS attacks are based on *algorithmic complexity attack* which were first introduced in [**?**] and formally described by Crosby and Wallach in [**?**].

The goal of algorithmic complexity attacks is to force an algorithm to run in the worst case execution time instead of running in average time. It has been employed against hash tables [**?**, **?**, **?**], quick-sort [**?**], packet analyzers [**?**] and file-systems [**?**]. For instance, in [**?**], hash tables are attacked to make the target run in linear time instead of the average constant time. This entails significant consumption of computational resources, which eventually leads to DoS. In Google Safe Browsing, we face a different problem: the data structure and the data representation used by Google, to be noted *Bloom filter* and *delta-coded prefix table* has a false-positive probability. Increasing the false-positive probability implies increasing the number of requests towards Google's servers made by honest users: safe browsing will be unavailable. To meet this purpose, we need to create pre-images and second pre-images for truncated hash function (cryptographic or not). We forge URLs with malicious contents corresponding to certain pre-image or second pre-image. Then, we use a more classical algorithmic complexity attack to make Google's servers overload its clients.

Google provides a Safe Browsing API for developers. A countermeasure to our attacks must be compatible with the current API, without any significant memory overhead. Our investigation leads us to propose Bloom filter combined with message authentication codes or universal hash function and full-hash description to thwart our attacks while achieving the best complexity.

After this introduction and related work, the paper is organized as follows. Section 2 first provides a description of Google Safe Browsing from the browser point of view. It presents

the data structures used and a simplified finite state machine for the behavior of the client. Section 2.4 is the main contribution of the paper and elucidates our two attacks. Section 3 presents the different possibilities to prevent our attacks, namely, randomization and prefixes lengthening.

2 Google Safe Browsing

Google Safe Browsing aims to provide lists of malicious URLs to prevent end users to access them. The service was started in 2008, and three different versions have been proposed since then.

Since the inception of Google Safe Browsing, other service providers have proposed similar solutions to detect malicious websites and inform the user about them. Yandex[1] provides a Google-type API compatible with C#, Python and PHP. The API has been used to implement the safe browser add-on developed by *Web of Trust*[2].

Microsoft promotes *SmartScreen URL Filter* [**?**] and is shipped with its products: IE, Outlook.com, Outlook, Exchange, Windows Live Mail and Entourage. Recently, Windows 8 extends the SmartScreen system to cover not only the URLs visited, but also files downloaded by the browser. The technique consists in a simple and direct *look-up* in a database of potentially maliciously links. In case of files, the hash of the file is sent to the server hosting the database.

Norton Safe Web (NSW)[3], developed by Symantec is another tool operating as a web browser plugin, and requires Internet Explorer 6 or Firefox 3. Similar to the SmartScreen Filter, the look-up is direct. Kaspersky restricts its Safe Browser product to low memory mobile devices such as phones and tablets. McAfee SiteAdvisor[4] is another product very similar to NSW.

2.1 General description

Figure 1 shows a simplified architecture of the service. Google Safe Browsing classifies malicious URLs into two lists: malware and phishing. These lists can be accessed by clients with two different APIs, software developers choose the one they prefer according to the constraints they have.

Google Safe Browsing 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 directly an answer for each URL. This is straightforward and easy to implement for developers but has drawbacks in terms of privacy and performance. Indeed, URLs are sent in plaintext to Google's servers and each request implies latency due to a network roundtrip. To solve these problems, Google offers another API: Google Safe Browsing API.

Google Safe Browsing API is more complex than the Lookup API and gives back to the client a part of the processing. To check if a URL is considered as malicious the client can not handle a URL directly like before. Instead, the URL is canonicalized following specification and digests are computed using SHA-256 [**?**], a cryptographic hash function. Digests are then checked against a database stored locally on the client which contains 32 bits prefixes of malicious URL digests. If there is a match, the client must query the remote Google service to get all the full hash values of malicious URLs for the given prefix. Finally, if the untruncated digest that the client wishes to check is not present in the list returned by the server, the URL could be considered safe. Figure 2 summarizes a request to the Google Safe Browsing API from a client's point of view.

[1] http://api.yandex.com/safebrowsing/
[2] https://www.mywot.com/
[3] https://safeweb.norton.com/
[4] http://www.siteadvisor.com/

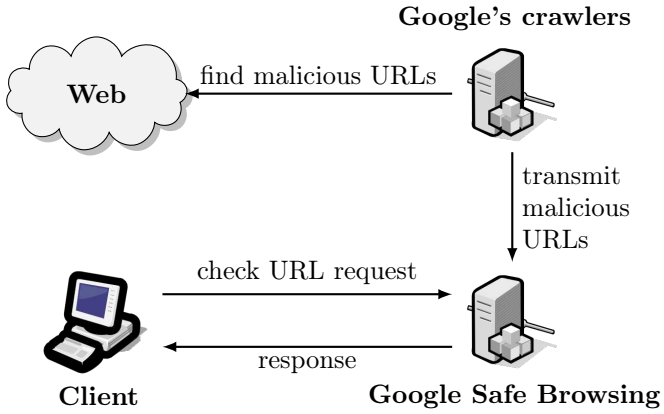


Figure 1: High level overview of Google Safe Browsing.

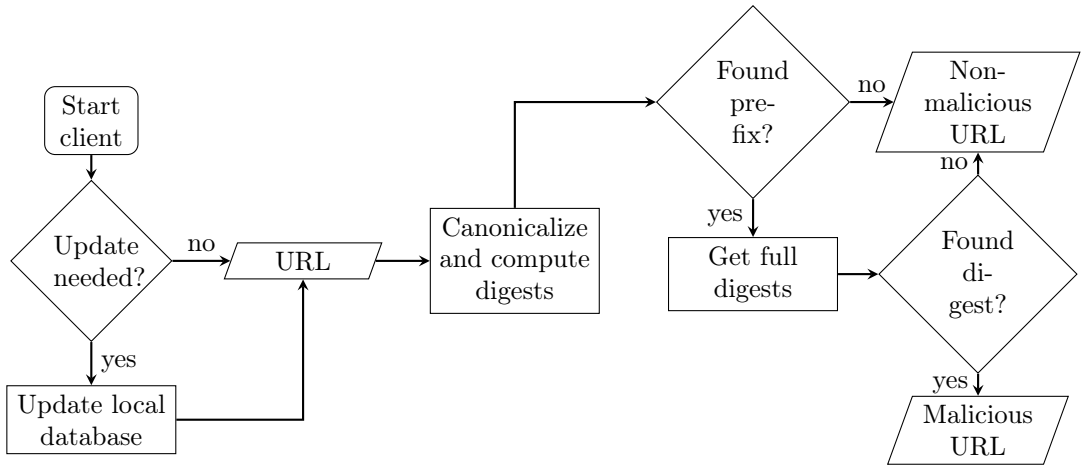


Figure 2: Google Safe Browsing client's behavior flow chart.

The prefixes database containing the blacklists is provided by Google and each client has to store it locally. Each blacklist is comprised of a series of *chunks*. A chunk is identified by an ID number and can contain new digests to add to the list or ID numbers of older chunks that have to be discarded. When a client updates its database, it sends the lists of chunk's ID numbers it already has and it gets back new chunks. In the same response, it retrieves the minimum delay it has to wait before its next update request. At the first start, the local database of the client does not exist and has to be initialized and updated. This may take a while until the database is fully synced and during this time, it is possible for a client to miss a malicious URL because it does not know yet a prefix. Note that, for now, the API gives the first 32 bits of SHA-256 digests to be locally stored but a client must be able to handle any length up to 256 bits.

After a match on the local database, the client needs to request the full list of 256 bits digests corresponding to the suspected prefixes fragments to ensure if the tested URL is malicious or not. The full digests is then stored instead of the prefixes until an update discard them. Storing the full digests can avoid further slow network requests.

In order to maintain the quality of service and limiting the amount of resources needed to

run the API, Google defined for each type of request the frequency of queries that clients must follow. Behaviors of clients when errors is encountered is also fully described.

2.2 Bloom Filter

Bloom filter introduced by Bloom [**?**] is a space-efficient probabilistic data structure which provides a solution to the *set-membership problem*, i.e. to check if an item belong to a predefined set. Essentially, a standard Bloom filter is a bit vector $\vec{z}$ of size m, initialized to $\vec{0}$.

Insertion The filter is built by inserting each of the items of a given set S into $\vec{z}$. An item $x \in S$ is inserted into a Bloom filter by first feeding it to k independent hash functions $\{h_1, \ldots, h_k\}$ (supposed to be uniform) to retrieve k hashes: $\{h_1(x), \ldots, h_k(x)\}$. These hashes are then reduced modulo m to obtain k bit positions of $\vec{z}$. Finally, insertion of x in the filter is achieved by setting the bits of $\vec{z}$ at these positions to 1.

Query In order to check if an item y belongs to S, it suffices to check if the item has been inserted into the Bloom filter $\vec{z}$. Achieving this requires the item y to be processed (as in insertion) by the same hash functions to obtain k indexes. If any of the bits at these indexes is 0, the item is not in the set/filter, otherwise the item is present (with a small *false positive* probability).

False Positive Probability The false positive probability occurs because of the collision on the reduced/truncated digests. If n is the number of inserted items into the Bloom filter, the probability of false positive is:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{1}$$

The optimal number of hash functions that minimizes the false positive probability is (see [**?**]):

$$k = \frac{m}{n} \cdot \ln 2 \tag{2}$$

We further note that the expected number of set bits in the filter after inserting n items in the optimal case is $m/2$.

Example 2.1 (Bloom Filter) *Figure 3 presents a Bloom filter of size* 12 *bits for a set of* 3 *items, constructed using* 2 *hash functions. A collision occurs on the first truncated hashes of* y_1 *and* y_3*. The item* $v_3 \notin S$ *is found to be present in the filter and hence is a false positive.*

Since its conception, Bloom filter has been used for various purposes in networking [**?**], and in designing efficient cryptographic primitives such as Searchable Encryption [**?**], Private Set Intersection [**?**] among others. In the domain of networking [**?**], Bloom filter are used for resource routing and are also employed to simplify or speed up packet routing protocols.

Usage in Chromium In an earlier version of Chromium (discontinued since September 2012), a Bloom filter was used to store the prefixes' database on the client's side. However, the false positive probability in this case is an issue because it could force clients to perform requests more often than necessary to get full digests for a prefix, eventually leading to a heavier load on Google's servers.

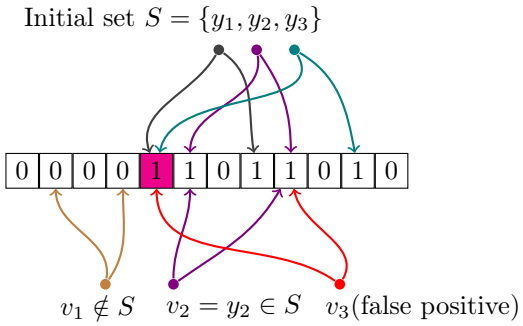


Figure 3: Sample Bloom filter using 2 hash functions. Items y_1, y_2, y_3 are inserted by setting the bits at the obtained indexes to 1, while items v_1, v_2, v_3 are checked for belonging. The colored cell represents a collision.

To minimize this, the Chromium development team used Bloom filter with 20 hash functions. The hash function used is the one proposed by Jenkins [**?**]. They also set the minimum size of the Bloom filter to 25000 bytes and the maximum size to 3Mb. The filter can then accommodate between 8000 and 1006632 items (on an average 25 bits per item). The false positive probability expected in this range can be computed using Equation 1. It is approximately 6.57×10^{-6}.

We note that this is not the optimal false positive probability, as for a filter size varying from 25000 bytes to 3Mb, and the number of hash functions being 20, the optimal number of elements that should be added lies between 6931 and 872180 (Equation 2).

2.3 Delta-coded Prefix Table

For the reason of false positives which may lead to unnecessary queries, and a significantly large memory usage, the Chromium development team have switched to another data structure in the more recent releases. Indeed, memory usage is a critical point for web browsers, especially when they are used in mobile environments. The current data structure does not have any false positive and uses less memory.

This data structure is based on *delta encoding*. Digest prefixes are sorted, and the difference of the consecutive prefixes is calculated. The final representation requires two arrays, stored using 16 bits. An index handles cases where 16 bits can not encode a difference and provides a quicker random access. A maximum of 100 differences between each index is allowed.

Example 2.2 (Delta-Coded Prefix Table) *Let us consider the following sorted prefixes:*

$$\{20, 42, 60000, 200000, 210000\}.$$

Two vectors are considered: index vector and delta vector. To store the prefixes into the delta-coded list, first the difference set is computed i.e. $diff = \{20, 22, 59958, 140000, 10000\}$. *A pair composed of the first prefix and the size of the initial delta vector i.e.* $(20, 0)$ *is then inserted into the index vector. Since, the subset* $\{22, 59958\}$ *of the prefixes can be encoded using less than* 16 *bits, the delta vector will now contain the list of these differences i.e.* $\{22, 59958\}$. *The following difference,* 140000 *however cannot be encoded using* 16 *bits, hence a new index is assigned, and the pair composed of the prefix and the current size of the delta vector* $(200000, 2)$ *is inserted in the index vector. Finally, the last delta is inserted in the delta vector since it can be encoded using 16 bits.*

This data structure has also a drawback: it is slower to query than a Bloom filter. However, it is still much faster than a network query and the balance between the execution time and the memory usage seems suitable with the operational constraints.

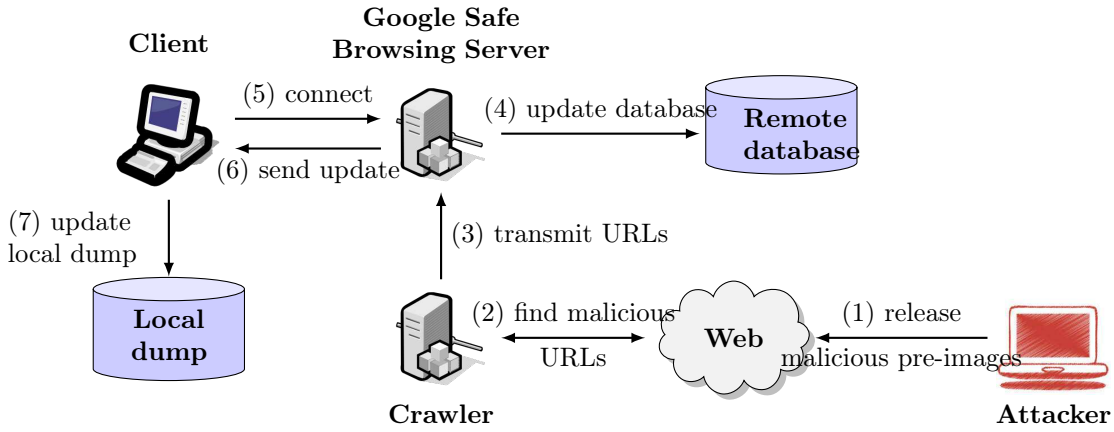


Figure 4: Attack on Google Safe Browsing: An attacker forges malicious URLs respecting certain constraints and diffuses them. These URLs then get detected and indexed by the Google crawlers and eventually reach the clients through the database update.

2.4 Attacks

We present a combination of pollution attacks and false positive flooding on Google safe browsing. Essentially the adversary targets the local data structure stored on the client side. Since in both the aforementioned data structures, hash function is the basic primitive, the core of our attack builds on the truncated digests available to the adversary.

We first describe the pollution attack on Google safe browsing followed by our false positive flooding.

2.4.1 Pollution attack on Google safe browsing

The attack can be divided into three phases: forging malicious URLs, including these URLs in Google safe browsing and the local database update (see Figure 4). In the following, we discuss these phases in detail.

Forging malicious URLs Let us consider a hash function H which outputs an ℓ-bit digest. Throughout the paper, we always consider that the inputs of the hash function are canonical URLs. Our goal is to attack the digest produced by the hash function. Non-cryptographic hash functions such as FNV [**?**], Jenkins hash [**?**] are designed to satisfy several criteria such as the avalanche test [**?**]. However, they are not designed to be resilient to an adversary as shown in [**?**]. In security, three attacks are considered against cryptographic hash functions.

- **Pre-image resistance:** Given a digest d, it is infeasible to find any input x, such that $H(x) = d$.
- **Second pre-image resistance:** Given an input x and the digest $H(x)$, it is infeasible to find another input $x' \neq x$, such that $H(x) = H(x')$.

- **Collision:** Find two distinct inputs x and x' such that $H(x) = H(x')$.

The choice of ℓ is critical for cryptographic hash function because the complexity for finding pre-image and second pre-image is 2^ℓ and $2^{\ell/2}$ for collisions. For large values of ℓ, *i.e.* the current cryptographic hash functions, this complexity is clearly too large. Truncated digests must be used carefully in [**?**]. In Google Safe Browsing, SHA-256 is used on the URLs but they are truncated to 32 bits *vs* 256 bits of SHA-256. With 32 bits of digest, pre-image, second pre-image and collision can be computed after 2^{32} computations. We naturally extend the notion of pre-image and second pre-image to *multiple pre-images* and *multiple second pre-images*: finding $t > 1$ pre-images or second pre-images.

Including our URLs in Google safe browsing After forging malicious URLs and malicious contents, the second step of our attack consists in including our URLs in Google safe browsing. There are three ways to do so. Either our 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 API to report our URLs as malicious using the following interfaces:

Malware:`www.google.com/safebrowsing/report_badware/`

Phishing:`www.google.com/safebrowsing/report_phish/`

Local database update Once these malicious URLs reach the Google Safe Browsing server, the server diffuses them by sending an updated local database to the clients in order to incorporate them. In this way, the adversary has established a data flow with the end users through Google.

2.4.2 False Positive Flooding Attack on Google Safe Browsing

Irrespective of the local data structure: Bloom filter or the delta-coded table, the adversary using pre-images or second pre-images may force the Google safe browsing mechanism to send queries to the distant server more frequently than required. This attack builds on top of the afore-described pollution attack. The impact of the attack depends on whether pre-images or second pre-images are found.

Pre-image DoS attack In case of Bloom filter, a pollution attack leads to an increased false positive probability. We note that the Bloom filter is polluted using malicious URLs generated during the first phase of the pollution attack. These URLs upon release finally gets included in the local database copy of each client through updates. The same holds for delta-coded prefix table. Due to the increased false-positive probability, the server is contacted more frequently than necessary. This consumes network bandwidth on both the client side and the Google server leading to a denial-of-service.

Second pre-image DoS attack Given a non-malicious URL, for instance the NDSS conference web page `http://www.internetsociety.org/events/ndss-symposium-2015/` and its 32-bit prefix `0xac47c1ec`, an adversary would exhaustively search for a second pre-image of the 32-bit prefix in the first phase of the attack. An illustrative example is `http://ndsss15isociety.webs.com/6177352392`. Such a URL (with malicious content) is then released on the Internet in the second phase. The Google crawler eventually detects the URL as malicious and adds it to its database. The prefix dump on the client side is accordingly updated to include the newly found URL. Since the malicious URL has digest prefix that matches the non-malicious URL, the

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

client is forced to make requests to get full digest for the concerned prefix, whenever he visits the actual conference web page. If digest prefixes of websites with large traffic are targeted, the number of queries can grow quickly and would consume the network bandwidth yielding a DoS attack.

This second pre-image attack works since GOOGLE Safe Browsing server does not seem to have any mechanism to whitelist web pages. The whitelist provided by the GOOGLE Safe Browsing API contains a single SHA-256 digest.

DoS amplification using multiple second pre-images Amplification attacks aim to magnify the amount of bandwidth an adversary can target at a potential victim. The original amplification attack was known as a *Smurf* attack [**?**], which involves an adversary sending ICMP requests to the network's broadcast address of a router configured to relay ICMP to all devices behind the router.

In order to mount a classical DoS attack against such web servers such as the ones of GOOGLE, a huge number of botnets would be required. However, using the technique previously described, an adversary would only need to generate/spoof t second pre-images of a URL. Upon releasing these malicious URLs, the adversary eventually forces a client to receive at least t full hashes, symmetrically forcing the GOOGLE safe browsing server to send this data. Consequently, network bandwidth on both sides are consumed.

We note this amplification attack also applies on pre-image DoS attacks, where multiple pre-images are found without necessarily attacking a specific URL. The number of these (second) pre-images defines the amplification factor of the attack. We also highlight that the query-only attack produces DoS with *reflection*, where the client unnecessarily receives 256 bits digest, while symmetrically the safe browsing server has to send them at each request.

2.5 Experiments and Results

We have written our pre-images and second pre-images search engine in python. Our brute-force-search being highly parallelizable, we exploit the module Parallel Python[6]. The domain name corresponding to each generated URL is also checked for its availability using the python module pywhois: a wrapper for the Linux `whois` command. The goal is to find unregistered and available domains and the URL paths on which malicious content could be later uploaded. All our experiments were performed on a laptop computer with CPython 2.7.3 interpreter. Our target platform is a 64-bit processor laptop computer powered by an Intel i7-4600U processor, with 4 cores running at 2.10 GHz. The machine has 4MB cache and is running 3.5.0-47 Ubuntu Linux.

In order to compare the robustness of the malicious lists provided by the safe browsing API, we considered 1M most frequently visited web pages and domains provided by Alexa[7]. 32 bits SHA-256 prefix of 43 of these domains were found be present in the malware list provided by GOOGLE, while 117 of these domains were present in the phishing list.

Query-only pre-image attacks require fake URLs generating false positives. Our URL search engine generated over 1 billion available domains and the corresponding URLs. Table 1 presents our findings on this list of potentially malicious URLs. For instance, we found 45 prefixes for which we were able to generate 7 pre-images.

We further observed that, the false positive probability of the chromium Bloom filter got doubled when only 60% of the total number of items were well chosen. Hence even if an adversary

[6] http://www.parallelpython.com/
[7] http://www.alexa.com/

Table 1: (Second) pre-image results on potentially malicious URLs.

#pre-images/prefix	**#prefixes**
7	45
6	1378
5	33359
4	640804
3	9813276
2	112750918
1	863692618

is not able to insert all the malicious URLs, she may still force the safe browsing mechanism to substantially deviate from its expected behavior.

3 Countermeasures

The design of a good countermeasure to our DoS attack must be easy to deploy, ensure compatibility with Google API and have reasonable time and memory size at the browser side. We first show how probabilistic solutions prevent efficiently Bloom's filter pollution but not second pre-image DoS and DoS amplification. The complexity of probabilistic solutions is clearly too important to defeat our second pre-image DoS and DoS amplification. The root of our attack being the computation of second pre-image, we advocate on solutions which increase the prefix length used to describe malicious URLs.

3.1 Probabilistic solutions

The first countermeasure proposed to defeat algorithmic complexity attack was to use universal hash functions [**?**] or message authentication code (MAC) [**?**]. Such methods work well in problem for which the targeted data structure is at the server side. Let us consider that we are dealing with a hash table or a Bloom filter without loss of generality. 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 it is not computationally feasible for an adversary to neither guess the function/key or pre-compute the attack for all the possible values. All the operations made on the data structure are computed by the trusted server.

For Google safe browsing, the situation is different. The original data structure is computed by Google and included in Chromium binaries. The data structure is then stored and used at the client side. It implies that no value, *e.g.* function or key, can be kept secret to the client. Two strategies are possible: universal hash functions are used locally by the client on the current SHA-256 URLs prefixes or the URLs prefixes are computed using universal hash functions. Each solution has an impact on the security and the complexity of the scheme.

When Chromium is run for the first time on the client, it loads the 32-bit prefixes into the data structure. If a Bloom filter is used, universal hash functions can replace the existing deterministic hash function. It results that an adversary cannot pollute massively the Bloom's filter of all the client around the world. However, it does not prevent second pre-image DoS and DoS amplification because the 32-bit prefixes are still used and they are obtained by a deterministic method. An adversary can still forge multiple second-pre-image to `google.com` for instance.

The alternative to the previous solution is to use universal hash functions directly on the URLs prefixes. The prefixes received are unique to each client. An adversary can make a second pre-image or DoS amplification only on a given user if he knows the function. But it can not be extended to other users. This solution defeats second pre-image DoS, DoS amplification and Bloom's filter pollution. However, it requires that Google recomputes the prefixes for each client.

All the browsers share the same function/key but this key is renewed by Google every week or month. An adversary can only pollute Google Safe Browsing only for a short period of time. In this last strategy, the Bloom filter must be re-computed on a regular basis by Google's servers with a different function or key and spread to all the browsers. It can represent a high communication cost.

3.2 Lengthening the prefixes

Table 2: Client cache for different prefix sizes.

		Data structure (MBytes)					
		Delta-coded table		**Bloom filter (google)**		**Worst case filter**	
Prefix size (bits)	**Raw data**	**size**	**comp. ratio**	**size**	**comp. ratio**	**size**	**comp. ratio**
32	2. 5	1.3	1.9		0.8		0.1
64	5.1	3.9	1.3		1.7		0.3
80	6.4	5.1	1.2	3	2.1	14.4	0.4
128	10.2	8.9	1.1		3.4		0.7
256	20.3	19.1	1		6.7		1.4

The case of Google safe browsing is different: we have to fix the client cache and the prefix system. The problem of the protection of the client cache is easy to solve.

The original data structure is not included in Chromium binaries. At the first execution of Chromium, the client cache downloads from Google servers the values to be inserted. It is thus possible for the client to choose its own local key to choose an universal hash function or a key for a MAC. In other words, we can apply the solutions discussed earlier to defeat pollution attacks.

Unfortunately, the same solutions can not be applied to protect the SHA-256 prefixes of the URLs. Indeed, the client needs to be able to recompute these prefixes. It is impossible for Google to share a global key or a secret with the client: it will be known by the adversary. A possible solution would be to make Google servers compute all the prefixes with a unique key shared with a given client. The key and the prefixes are then sent to the client at the initialization of Chromium. The drawback of this solution is the load for Google servers because they must share and manage the keys with the clients and use them to recompute all the prefixes. This solution does not scale well and is indeed impractical.

The core of our attack is the computation of pre-image and second pre-image. 32-bit prefixes are not enough to prevent those attacks and thus increasing their length is the obvious choice. Unfortunately, it has some effects on the size of the data structure at the client side. We consider two choices: delta-coded tables and Bloom filters. From Table 2, we observe that delta-coded table does not scale well if we increase the prefix length. For 64-bit prefixes, the size of the data structure gets tripled. For the acceptable cryptographic security of 80 bits, delta-coded table is around 5MB. While using 256 bits of the prefix, a client needs to interact with the server only once for the purpose of storing the data structure. Any later queries can be locally performed. This can also be seen as the trivial *private information retrieval* protocol. However, this mechanism would increase the size of delta-coded table to 19 MB. The size of the Bloom

filter remains immune to the change in the prefix size but unlike delta-coded table, the data structure would no longer be dynamic. The size of the worst case filter is 14.4 MB compared to the 3 MB of the original one. Increasing the size of the prefixes has no effect on the parameters of the Bloom filter. This is the solution we favor.

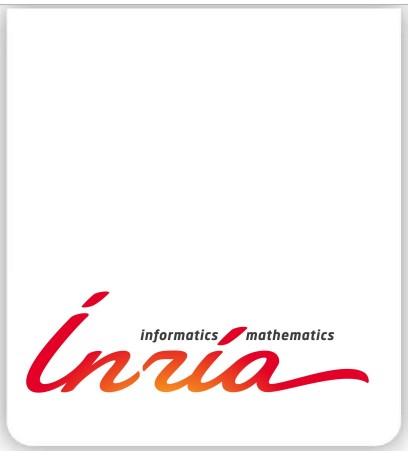
4 Conclusions

We have described several weaknesses of Google safe browsing to mount DDoS attacks against Google's servers and safe browsing users. In Google safe browsing, Google attempts to implement a cache at the client side to reduce their servers load. The design of this cache leads to use weak data structures. The digests are too short digests or the Bloom filter was unsafe. Increasing the digests length is possible but the current data structure of Google does not scale well with this modification. Bloom filters are oblivious to such modification and combine universal hash functions or MAC it offers a safe solution.

We proposed also a new direction for algorithmic complexity attacks based on false probability. To our knowledge, we are the first to saturate/pollute Bloom filters. It is almost certain that our strategy of attack against Bloom filters can be replicated to other contexts: Bloom filters are widely used in networking and security. It is critical to know if a adversary can control or not the content of the filter.

Acknowledgement

The work was also partly supported by the LabEx PERSYVAL-Lab (ANR-11-LABEX-0025) and the project-team SCCyPhy.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399