University of Amsterdam

System and Network Engineering

Security of Systems and Networks

Exhaustive Search on URL Shorteners

Alexandros Stavroulakis
Alexandros.Stavroulakis@os3.nl

Xavier Torrent Gorjón
Xavier.TorrentGorjon@os3.nl

Nikolaos Petros Triantafyllidis
Nikolaos.Triantafyllidis@os3.nl

December 20, 2014

Contents

Abstract

NOTE TO TEAM: This is just a first attempt on an abstract that can work as a guiding light. We'd better write the abstract after the report is finished. Which makes more sense. Peace. And love.

In this project we focus on URL shortening services, from a security point of view.
Our first aim is to determine the feasibility of an exhaustive mapping of all the short links to their respective long URLs, estimating the cost in both time and computational resources. Secondly we try to discover the nature and the amount of sensitive (usernames, passwords, system configurations, user details, etc.) data that has been deposited to such services, and eventually pinpoint security holes that might have been leaked through them. Our final aim is to try and determine if there is some sort of mapping relationship between the long and short URLs. The research methodologies and software tools used for the project are described in detail. The results and interesting findings are presented and the appropriate discretion is applied where deemed necessary.

Acknowledgements

At this point we would like to thank KPMG and especially Mr. Martijn Sprengers and Mr. Ruud Verbij for their assistance to the completion of this project.

1 Introduction

URL shortening refers to the technique of taking any HTTP Uniform Resource Locator (URL) and producing a shortened version that links to the same Web resource, by issuing an HTTP redirect response. The purpose of this technique is to transform large (sometimes hundreds of characters long) and very descriptive URLs to something that is much sorter, easier to remember and be shared in an environment where typing space is limited (social media, mobile devices, instant messengers, etc.)

This technique has been around since the early 2000s, when the first US patent was filled, but became really popular by the coming of Twitter, a social medium that only allowed a certain number of characters to be typed in each post (Tweet) of the user, and which started automatically shortening URLs more than 26 characters long. The first website to provide shortening services was tinyURL.com, with similar services including, among others, wp.me (by Wordpress), goo.gl (by Google) and bit.ly, with the last two being the most popular.

This report focuses on certain security issues that arise by the use of such services. The main problem is that the databases kept by these services in order to issue the correct redirects (from short to long links) are public and easily accessible via a simple web client. In case the users of these services carelessly input sensitive data, this data becomes public and easy to retrieve by anyone.

The rest of this chapter is dedicated to describing in detail the problem we will be examining, presenting the previous work on this domain and mentioning certain ethical implications that arise from our study. The second chapter is a description of the URL shortening methods in general and the two services that have been used in this study (goo.gl and bit.ly) in particular. The third chapter presents the research methods and software tools that we have designed, developed and used in this project. The fourth chapter demonstrates the results that have been produced by our research and the security implications that arise in terms of user privacy and system security. The next chapter is a discussion about suggestions and solutions that could help mediate the security problems of such services. The last chapter summarises the conclusions of the project and proposes ways to improve the current work.

1.1 Problem Description

URLs tend to carry a lot of information besides the location of the web resource. That can include, among others, file hierarchies, configuration parameters, IP addresses, port numbers and in more serious cases, usernames, clear text passwords, links for unauthenticated access to internal servers, etc. Moreover, URLs can link to web content that would normally be inaccessible by non-authenticated users, through permalinks and hotlinks. Namely that includes social network profiles of users, private pictures and videos, online documents etc. It is apparent that there is a lot of web content that can either be exploited for attacking the computer infrastructure of companies and organisations or can lead to user identification and leakage of their private data. Obviously, in all of the aforementioned cases this content has to be securely kept away from the public eye.

Hunting for URLs and web content that can be exploited is not something new. Scraping the web for exploitable data has been around since the early days of the web. Since the Web is the most popular among internet users, "hackers" are always trying to find content that users share in private. For example through a technique called "Fuskering" it is easy to guess the address of certain web content by following a specific pattern. The problem, however, in the previous cases is to be able to locate the source of the information to be retrieved which sometimes needs to breach the security of certain systems such as emails, tcp connections, etc.

In the specific case of URL shortening things become much easier. First of all a potential attacker has a very well known and centralised source of information since URL shortening services expand their databases each time a user shortens a long URL. Secondly, contrary to long URLs that can be highly unpredictable, containing extended file hierarchies, non-standard names, etc, shortened URLs are very formalistic drawing from a limited character space and having a limited length of a few characters. That allows anyone to try and guess random URLs and be redirected to their full length counterparts. If that effort gets co-ordinated and spanned across various machines, one could exhaustively search across the domain of the short URLs or at least get a very big portion of the long URL data space, on which they could easily search for patterns that may expose sensitive data. A third "charming" attribute of the URL shorteners is that the information collected is very diverse, spanning from blog posts to internal network configurations, making this source very interesting to any person digging for security related information.

The problem becomes even more intense when, not only users unaware of the public nature of a short URL give away sensitive personal information, but also applications that make use of the public APIs of the shortening services automatically shorten long URLs that their users share. In that case a user might indeed have shared a long URL only with their selected contacts in a private environment which at the same time gets exposed to the public through the shortener service.

1.2 Previous Work

Most of the previous work which has been done on URL shortening services is based on the use of short URLs and its correlation with SPAM and phishing techniques, whether that is to prevent spamming or to explain how these two are combined. One example of the latter is how the original URL is masked in a way that the receiver of such a malicious email will not be able to realize the fact that by clicking such a link, he or she will not be redirected to a legitimate website.

Another example was the investigation of specific countermeasures taken from these particular services to defend against the manipulation of the shortened URLs for malicious purposes; also trying to determine and statistically analyse the extent of spamming given certain geographical locations in which the services were used.

As for the analysis of the URLs as independent links and their respective data, the primary focus was on short URLs collected by popular social media such as Twitter. And the statistics revolved around their popularity lifetimes along with the expectancy of the amount of clicks these URLs would get.

An interesting example of this was to use these services to monitor certain campaigns from companies that someone deems as competitors. Google search queries were used to find results from the website of a specific company that contain the names goo.gl or bit.ly in combination with the use of certain keywords. After obtaining a positive result, one could use these shortened links to study their metrics and then judge (e.g. by the amount of clicks) how well that particular campaign is going.

The most closely related to this project scientific research was a paper published by the RWTH Aachen University in 2010. This paper, among other security related aspects of the URL shortening services, examines a potential enumeration of all possible short URL strings. However, they have tried a much different approach, only taking a small sample (around 230000 URLs per service) of the possible space in a time amounting to a few hours for that process (they mention for example 27 hours for 230000 URLs from goo.gl). Based on that they conclude that an exhaustive search is feasible without, however, justifying that conclusion.

1.3 Ethical Implications

In any security related research, it is very likely to come across very sensitive data that has to be treated with care in order not to fall into the wrong hands. Any findings with particular security significance will not be disclosed in public and if needed the appropriate parties will be notified. Any data collected during this research will be kept safely in University infrastructure, and appropriate measures will be taken to prevent leakage. It is however useful to note that all data gathered are easily retrievable by anyone since they are public data.

2 Shortening Services

Each long URL that gets shortened is associated with a unique ID, which is the part after the top-level domain name, for example the short URL http://bit.ly/1ypDUGk has a ID of 1ypDUGk.

Each time a short URL is followed by a web client the corresponding shortening service issues a HTTP redirect response. The redirection instruction sent to the client can contain in its header the HTTP status 301 (permanent redirect) 302, or 307 (temporary redirect).

There are several techniques to generate unique IDs. Those can be encoded in base 36, assuming 26 letters and 10 numbers or alternatively, if uppercase and lowercase letters are differentiated, then each character can represent a single digit within a number of base 62 (26 + 26 + 10) and each. Each ID can be generated sequentially following the order from 0 to 9 and from Aa to Zz. Another way would be to invoke a hash function or a random number generator so that the ID sequence is not predictable. Some services allow users to propose their own keys. For example an authenticated user of bit.ly could shorten the URL https://www.os3.nl as bit.ly/ohesthree.

The most popular of these services record inbound statistics that are publicly accessible by just adding some extra information at the end of the shortlink (such as the '+' symbol). These statistics include number of clicks, the countries where the link was accessed from, client platforms, etc.

Some shortening services pose restrictions on the format of the URLs to be shortened, that including the protocols supported for shortening. Others seem not to bother with checking the format of the URL, with goo.gl in particular being able to create shorthand links even for plain text, so long that it does not contain spaces.

The services chosen for this study were bit.ly and goo.gl mainly because they are currently the most popular services and also because they expose simple RESTful Web APIs through which we could obtain the data for our research. More details about these services functionality follows.

2.1 Goo.gl

The Google URL shortener known as goo.gl appeared in 2009. It was first only available through Google products such as Google toolbars or Google Chrome extensions. Since 2010 the service become accessible via a direct interface.

The short link IDs generated by the service are 4 to 6 characters long and are Base62 encoded. The user can append the '+' at the end of each link and instead of being redirected to the actual long URL they are presented with the long URL in text form and a number of statistics including number of clicks through history, referrer countries, browsers and platforms used. There is also a QR code generated that directs to the same long link.

The service can be accessed programmatically via the Web API that gets exposed by Google. The API is available among most of Google APIs via the Web-based developer console. In order for it to be used the user must have a valid Google account and register their application at the developers console. Then the URL shortening API must be activated and an API key must be generated. There is an option for OAuth token authentication but for this particular API most operations are available with plain API key authentication. The usage for the URL shortener API is limited to 1.000.000 requests per day and the default limit per second is set to 1 request/s. This however can be configured via the developer console to thousands of requests per second.

The API exposes 4 different functionalities which are the following:

- Shorten a long URL: This functionality allows the user to shorten a URL programmatically as they would do it from the webpage of goo.gl. The request must be made with the http POST method and provide the long URL as a json formatted string. The output is also a json struct. The documentation recommends the use of a API key for this operation. If however an auth token is provided the short URL will be unique. Otherwise it might be reused from a previous request to shorten the same URL.
- Expand a short URL: Users can call this method to expand any goo.gl short URL. The request is directed with the http GET method and

the input is given as a URL parameter. The output is given as a json struct. The documentation recommends the use of a API key for this operation

- Look up a short URL's analytics: Much like appending a '+' at the end of a short URL the analytics of a short URL can be accessed programmatically via the API. The user must first issue an expand request like the one described above, appending an extra URL parameter to add for details. The analytics along with the long URL are returned in json format. The user can also filter only the analytics fields they are interested in by adding extra parameters. The documentation recommends the use of a API key for this operation.

- Look up a user's history: This request fetches a user's URL shortening history. The data that this method returns are the same as those returned by an analytics request. This method must be authorised with an auth token.

2.2 Bit.ly

Bitly offers a URLS service that reduces long URLs to a bit.lu URL with a hash that ranges from one to seven characters. This gives the platform a considerable higher amount of addresses than goo.gl.

The bitly API has a good documentation of its possibilities and how to use them. However, some important details are omitted -as an example, the documentation states that there are many request limits per amount of time, but does not specify the actual limits-. These additional options can prove to be useful for more complex attacks, and we will discuss the possibilities of the most interesting ones.

- Expand: The basic functionality of providing a short URL and retrieving its related long URL

- Lookup: Returns a bitly short link, provided an URL. Its an analogy of a reverse DNS query

- Info: Returns all the information associated to a bitly URL. This includes its related URL, as well as other interesting parameters such its creation time or the creator of that link. This is useful as this kind of information cannot be retrieved through the front-door of bitly

- Shorten: This function produces a new short URL, in a similar way as the user interface provided in the website. Although less interesting than the others from the point of view of an attacker -as it is not an attack point-, this function accepts an additional string parameter to manually modify the generated short URL, effectively meaning that if a certain user or company uses this method carelessly, they can be easily targeted.

3 Research Methodology

In order to perform a mapping between the short and long URLs, we developed simple software to perform the expansion of the URLs, aiming to retrieve a subset of the original URL information stored on those services. The languages chosen for that operation are Go and Python for the goo.gl and bit.ly shortening services respectively.

To that end we used the public web APIs provided by those services. From the functionality exposed through these APIs, the URL expansion capabilities were only used. It is worth mentioning at this point, as will be discussed in later chapters as well, that the use of the public APIs to retrieve the data is not mandatory as one could call the short URLs and collect the redirect repsonses. The reason behind the use of these APIs was to follow the path recommended by the services, that would grant us enough data to conduct our research. High traffic directed to the "front door" of the services, originating from a single IP-space, could be presumed as an attack and lead to trouble such as rate limiting, blocking or even legal issues that could involve the University of Amsterdam, something which is of course highly undesirable.

The data gathered were stored both in flat files and a no-sql database, namely a MongoDB instance. On the collected data targeted patterns were search through plain regular expressions and data aggregations on MongoDB.

3.1 URL Expansion

Both services expose Web APIs that allow easy retrieval of the URLs stored on their databases. Moreover they provide other interesting features, like additional information about these URLs such as creation times, number of clicks, etc. Those features were not used directly on this project, but can still provide interesting security information if handled correctly.

The APIs limit their usage according how many calls they allow to receive from a certain authenticated user per some fixed amount of tim. The goo.gl public API has a generous limit of 1.000.000 calls a day per API key. The amount of requests per second is by default set to 1 but this can easily be configured to tens of thousands via the web based developer console. As for bit.ly, the only limit stated in its official documentation is that of 5 concurrent connections. There is however a significant hoURLy limit, of about 4,000 requests. This value is not stated on the documentation website, so we just calculated an approximation. At the end of the designed week for data collection purposes, we had retrieved 7 million unique, non-empty URLs from goo.gl and 3 million from bit.ly.

3.1.1 Goo.gl

For the expansion of the URLs shortened by the Google URL shortener, we developed a simple program in Go. This language was chosen for its speed, rich Web functionality and very strong concurrency model.

Since the goo.gl service uses Base62 encoded short URL IDs 4 to 6 characters long and we are bound by the usage limitation of the API (1.000.000 requests per day), each run expands $62^3 = 238,328$ short URLs per run. To achieve this the user initialises the execution of the program with a constant prefix, 1 to 3 characters long and the software exhaustively searches through every Base62 combination for the remaining 3 characters.

The software utilises the concurrency model of Go which consists of very lightweight processes sharing the same address space called ”goroutines”. Due to the nature of this model thousands of goroutines can run concurrently, with minimal overhead for the system.

Each run takes 16 minutes to complete on average, making nearly a quarter of a million requests to the Google URL Shortener API. This allows the program to be run 4 times a day utilising slightly less than the daily rate limit. That means that within an hour we can gather nearly a million long URLs. At the end of each run the gathered file gets filtered with the use of traditional text editing tools to remove empty lines and lines that contain error messages as well as duplicate URLs (since goo.gl may shorten the same long URL with various different short IDs). That effectively gives us around 200.000 unique URLs per run.

Since Go is able to span tens of thousands concurrent goroutines and although these processes have a very small memory footprint, in our case each concurrent process makes an http connection that uses a separate file descriptor meaning that thousands of files must be open at each time. This can lead to various problems, especially if the file descriptor limit of the machine is very low. In our case we took various measures to prevent this. First of all the file descriptor limit for each machine was set to 99999. Secondly, although no synchronisation is needed between the goroutines that are autonomous, a semaphore wes set to block the execution of the program every until a group of 1922 routines was finished. The error handling system of the language was called in order for the execution not to be disrupted in case some exception occurred. In that case an Error message was printed. For the same reason the goroutines did not write to the output file directly, rather they printed to the standard output which was then redirected to the output file. Of course these do not reflect the best software development practices but due to the limited amount of time for this project quick fixes were called in order for us to have a significant sample of the dataset in short time.

3.1.2 Bit.ly

For the bit.ly crawler we used a more standard approach using regular parallel computing methods available on the Python Standard Library, in conjunction with the bit.ly API for Python. The priority for this project was to develop a very modular program (as bit.ly hoURLy limit was low enough that performance didn't really matter) so that the code could be expanded to allow for more complex utility such as also downloading HTMLs or documents from the retrieved URLs.

The first versions of this crawler used new processes for the parallel computing. However this caused some issues that were difficult to fix in the short time we had for this project, and we decided to switch to threads. In Python this can be a huge setback, as the Threading library runs all generated threads on the same processor, but in this environment the processing speed was far from being a bottleneck for the performance, compared to the network communication time between our clients and the server.

Another issue we had to fix was to make the code exception-safe and thread-safe.

- Safety on exceptions: Exceptions while doing the URL expansion were quite common and they needed to be handled correctly, as otherwise threads could die or start unpredictable behaviors. Exceptions ranged from encoding problems due to strange characters on URLs (Arabic or Japanese characters, for example) to queries returning errors (such as timeouts or reaching the limit set for a time range).
- Safety on threads: Threads worked independently from each other, but they were using the same output files for their results, to prevent excessive usage of memory and keep the data organized. This constant writing to the output files from different sources had to be managed carefully to avoid data corruption.

We also included a basic network communication utility on the script to enable communication with other machines. Although this was solely used to control the script remotely, it could be also used as a Proof of Concept on how an attack with a botnet could be used against those services to get all their database in a relatively short amount of time.

3.2 Data Extraction

Once the URLs have been retrieved, there are many ways that information can be handled, and different levels of depth to perform searches.

3.2.1 URL Data

The first level of search is the long URLs retrieved themselves. Some of those URLs contain parameters which can provide useful information to an attacker. Even a simple check for the string "password" leads to many URLs that actually have plaintext passwords on them. Other parameters that an attacker could be interested in are for example usernames, dates or session identifiers.

3.2.2 HTML Documents

A deeper search can be done by retrieving the actual HTML or documents behind the URLs.

- HTML mining: HTML pages can contain information poorly defended against attackers that can see or guess certain identification variables on the URLs to get extra data from the pages or their databases. If these identifications or sessions are not correctly handled, an attacker can view the same HTML page a legitimate user saw before him and obtain private data from it.

- Document mining: Some of the URLs we crawled were direct links to documents or FTP servers. It is easy to retrieve information from those sources in a similar way as the HTML mining, and obtain documents that should not be of public domain. Many of these URLs lead to public information that has no value for a potential attacker, but unawareness of how URLS work might lead to sharing private documents through them, which can be later retrieved if these links do not expire.

3.2.3 Regular Expressions

After discussing what can be done to search for and access information in the collected data, we can continue to the way we performed those searches and the results we obtained. It should be reminded that since goo.gl does not check if a long URL has already been shortened, we observed that the same URLs had been shortened more than once.

Some the of the keywords used to reveal sensitive information have already been hinted above. This is the list of our searches along with what we were hoping and expecting to find.

- **Password, pass or passwd.** Our first search was to see if we could possibly discover clear text passwords or even the hash of them that could be used in a replay attack.

- **Username.** We were hoping to discover user credentials that were sent over the line without being encrypted to protect them.

- **Mail.** In combination with the above, we were interested in seeing all types of user information that could be collected.
- **Admin or administrator.** Similar case with the aforementioned username. However we were hoping to obtain information that would lead to parts of websites which should not be accessed by simple visitors.
- **Login.** Usually in combination with the above. Also to see whether or not we could record session IDs.
- **Maps.** We were already assuming that many of the results would be location related. Our goal was to investigate whether we could find exact coordinates shared online and what information they would give us if we combined it with the analytics offered by the services such as creation times.
- **Common service names.** We searched for keywords such as mailserver, webserver, PhpMyAdmin, MySQL etc.
- **Ftp.** Our main interest in this was to test whether we can gain access in malconfigured FTP server, just by editing the long URLs and deleting the parts of the files that were shortened and shared.
- **Exploitable Information.** URL links with the file extensions like .php, .jsp, .aspx etc can contain information which could be exploited.
- **Cgi-bin.** Shared URLs which lead to cgi-bin scripts.
- **Shadow.** Shadowed password files that could expose some system vulnerabilities.
- **.ini** Configuration files of certain platforms or software that were shared.
- **Network Gear Vendor Names.** We searched for keywords such as Cisco, Juniper etc. We were expecting to find pieces of information from logfiles, maybe some firmware update indications.
- **Company/Bank/Institute Names** The same idea as above. We were hoping to maybe find confidential files or links that would give access to badly configured servers of these companies

- **IP addresses and ports** Since there are a lot of testwbesites that do not bother with a domain name, we were interested in seeing what type of information we could retrieve from this which could also lead to network and port scans.
- **goo.gl or bit.ly.** A fun search was to see if people tried to shorten already short URLs.

An example of the above searches is the following grep command with the regular expressions of filtering long URLs with IP addresses in them:

```
grep -E "(http://[1-2]0,1[0-9]2[.][1-2]0,1[0-9]2 [.][1-2]0,1[0-9]2[.][1-2]0,1[0-9]2)"
```

Using the aforementioned keywords, we were able to discover quite a lot of interesting information with some sensitive data which of course for confidentiality reasons can not be mentioned in this report. We do have the intention of contacting the related parties and notifying them about these privacy and vulnerability issues so they can take the seriousness of the matter into consideration and maybe find a solution to this issue.

3.2.4 MongoDB Queries

Although no actual work was performed on MongoDB the possibility of using a NoSQL database system like MongoDB was examined. This can account for scalability once the amount of data gathered grows in size up to a point where conventional operations are not efficient any more. MongoDB also offers certain aggregation frameworks and Map/Reduce operations that can help apply more sophisticated pattern matching and analysis techniques in the future.
An example query for searching the keyword 'password' in the dataset would be the following:

```
> db.collection.find({"long": /.password./})
```

One could also use the Aggregation Pipeline framework to spot duplicates and aggregate all the short links that correspond to the same long URL into one line. That way the analysis for ID generation patters could be aided. One example is the following:

```
> db.google.aggregate([{$match:{}},
{$group:{_id:"$long", short: {$addToSet: "$short"}}}])
```

4 Results

We did some basic mining on the data we obtained, which lead us to a large amount of vulnerabilities that could be easily exploited. Considering that we only did this very basic study on the surface of the possibilities available, and the fact that we only crawled a very small subset of all available URLs on these services, we can conclude that the URLs are a huge database of vulnerabilities on the Internet.

4.1 Research Findings

With a basic study of the data retrieved (meaning that we believe there could be many more issues to be exploited there through the usage of more complex tools) using the methods described in section 3.2, we found out the following problems:

- **Username and login pairs embedded as plaintext in URLs:** We found many instances of this issue from various sites, some of them being more dangerous as the usernames made references to email accounts or personal identification numbers, which could be used to track them down and exploit other services those persons use. Moreover, administrator information was shared, sometimes providing access in parts of websites that are not meant to be seen and accessed by the public. However, most importantly we discovered services that contained passwords in clear text or just the hash of the password and could grant access to accounts. In combination with the email accounts and and username information, it could become quite easy for an attacker to use this information for other services these users use and (considering the fact that many people use the same username and password pairs for different websites) access the rest of the services they use. Lastly, quite a few occurances contained password reset links, many times from the same websites.

- **Google maps location and timestamps sharing:** Some applications use the URL services to share information about a person visiting, for example, a certain café or shop and publish them in limited-length services such as SMS or Twitter. As the URL shortening services can offer information on when a link was created, this can lead to a violation of a person's privacy, by providing information of where they were at a certain hour, even if these applications do not provide that information.

- **Retrieving unprotected files from websites or FTP servers:** During our research we decided to search for certain types of files on the URLs, such as Microsoft Excel files. That was one way of investigating these systems; the other way was editing the long URL itself and trying to access different directories that would lead us to separate sections of the FTP server and maybe to more sensitive information. We managed to retrieve many unprotected files from FTP servers that had information about customers such as telephone numbers or addresses, and we even managed to find account balances. Also exact IP addresses and port numbers that are used to access them. This constitutes a major vulnerability that can be directly exploited by an attacker without any further effort.

- **IP addresses and ports:** We were able to find a lot of IP addresses shared and many of them in combination with open ports of the webservers.

- **Common Service Names:** Some of these services just linked directly to a login page of a mailserver or webserver. We discovered links to PhpMyAdmin or MySQL pages. This does not constitute as big a vulnerability as the above, but an attacker would still have the option of attempting to brute force these pages.

- **Network Gear Vendor/Software Company Names:** Some of the results we found were logfiles of certain routers after firmware updates, or antivirus database updates which could also lead back to an FTP server. However that mainly lead to the public section of the FTP server.

- **Company/Bank Names:** We are glad to say that the majority of the results we found from banks did not contain sensitive information. We did however witness some companies sharing internal links to systems that are solely aimed to their staff.

- **HTML scraping:** During our research we didn't have time to go in depth with this kind of attack, as it is more complex than the others and requires extra time and resources to be performed. However, from the URL expansion we could see many URLs having session identifiers on them that could lead to data leakage, meaning this kind of attack could be proposed as future work for in relation to this project.

4.2 Attack Options

In this section we will discuss our calculations regarding the feasibility of getting the entire space of URLs available on these services, as well as what can be done with that data.

4.2.1 Retrieving the Entire URL Database

First of all, we wanted to make a calculation on how much time would an attacker need to get all the URLs -or most of them, as it is a database constantly growing up-. Our size estimations are as follow in Table 1.

	Goo.gl	**Bit.ly**
Address Space	62^6 directions = 5.68 x 10^{10}	62^7 directions = 3.52 x 10^{12}

Table 1: Address Space for the goo.gl and bit.ly services.

A potential attacker could use a botnet to retrieve addresses. A botnet attack in this scenario can be useful for two purposes:

- First, the obvious advantage of retrieving URLs faster if the botnet is configured properly.
- And second, but not less important, it becomes a lot more difficult for the URLs companies to detect the collecting and expanding process and take prevention actions.

It is difficult to define a botnet size for this study, as botnets have a huge discrepancy in sizes, ranging from a few thousands to multiple millions of infected computers [1]. For this reason we decided to base our calculations on a botnet of a relatively small size, about 10^4 computers. This should be easily attainable worldwide, but even for a small region the number is reasonable.

The only parameter that we need to know after defining a botnet size is how many requests can we get per second. Assuming that we have a botnet that operates worldwide, we can estimate that the response times will vary from a few milliseconds to a few hundred milliseconds.

We show a brief study we did on this matter in Table 2, using information available on nirsoft.net[2] and RIPE Stat[3]. We selected various machines on different locations to get an estimation on the request times.

Server Location	**Minimum**	**Average**	**Maximum**	**Std Deviation**
Netherlands [4]	7.254	7.316	7.439	0.035
Japan [5]	257.859	265.058	269.539	4.787
USA (MI) [6]	105.340	105.389	105.479	0.436
Spain [7]	47.609	47.859	48.671	0.366

Table 2: Average roundtrip times to various countries from the SNE laboratory. The time is in ms.

From this data we can assume a request will take somewhere between 10 milliseconds and 300 milliseconds (we are sure we could find cases worse than the ones presented but that data already gives a good estimation on the relation between distance and time). However, considering these services are most probably replicated in multiple places around the world, we can expect the average request to be on the range of 100 milliseconds or less.

Considering that these computers could make parallelized requests, we estimate that running 30 low-memory threads is a reasonable value (the data transaction between the bots and the URLS servers could be limited to just the request URL and the long URL retrieval, making each request length about 20 bytes and each response had an average of 100 bytes for goo.gl and 110 bytes for bit.ly, not considering the length of TCP/IP or HTML headers).

All the presented data up to this point is synthesized as follows:

- For goo.gl URLs, where we had a address space of $5.68x10^{10}$, we can reduce that number to 5.68 x 10^6 by considering the botnet of 10^4

computers, assigning a different range of addresses to each infected computer. Each computer works with a dataset of 5.68 x 10^6 addresses, and considering they run 30 threads at a speed of one request per 100ms, we get up to 300 requests per second. The conclusion for this calculation is that it would take 1.9x10^4 seconds to get the entire database of URLs, which translates to roughly 5 hours and 20 minutes.

- Following the same procedure as with the goo.gl case, for bit.ly we end up with a total time of 328 hours and 30 minutes, which means expanding the whole set of data would take almost two full weeks (14 days).

These times mean that it is feasible to get the whole database of both services in a reasonable amount of time (specially goo.gl, which takes just a few hours). We also need to consider the fact that attacks can be started with just a portion of that set, and that these times can easily be further decreased by increasing the botnet size or trying to localize the infected computers on a certain location close to the URLS servers.

4.3 URL Creation Patterns

One of our main research goals was to discover if there exists any type of pattern in the creation of these short URLs in relation with the given long URLs and try to identify it. In the time we devoted in this research, we believe with a fair amount of confidence that these links are created almost entirely randomly from goo.gl. In the course of these four weeks, each time we tested this particular service we would receive a completely unrelated outcome to the previous tries.

It should be noted that the only indication of sequentiality in goo.gl is the fact that the length of the URLs grows as time progresses. From the data we collected, we can safely deduce the fact the four character long URLs (which is the minimum length) were created earlier than the URLs with length of five and six characters. Still the combination of characters used in each short URL seem to be chosen completely at random.

However that is not the case in bit.ly. During our research we observed that each time we would shorten a URL which had never been shortened in the past (e.g. links from our own Wikis), we would receive a link that would start with "1y[xxxxx]", where "x" represents a random character. By the end of our research, this value had progressed to "1z[xxxxx]", which is to be expected considering how much these services are used nowadays all over the world.

We believe that this issue may have to do with the fact that bit.ly keeps a database of all the URLs which have been shortened, always providing the same short link for the same long one. When instead, goo.gl -at least from the front end- each time we tried to shrink a long URL, we would receive a different result.

5 Suggestions

After studying our research results and having witnessed what kind of options and possible offensive routes are available for an attacker to choose from, we have thought of some helpful suggestions that could improve the user privacy issues shown above and also help avoid careless mistakes which could expose system vulnerabilities.

5.1 Awareness

First and foremost, we strongly believe that most of the issues with User Privacy could be avoided or at least be reduced by a great amount if the users were aware of the fact that the URLs they choose to shorten are and always will remain public information for anybody to collect and access.

While Goo.gl states in its webpage "All goo.gl URLs and click analytics are public and can be accessed by anyone.", Bit.ly does not mention anything in particular that could warn the users of the potential risks of sharing something personal without a real understanding of how the service works or for how long it will be available for the public.

Heavily used services like these need to help users protect their privacy and inform them of how they operate with clear and precise statements. Of course the users need to also be able to comprehend the actual meaning of the word public and realize the implications and consequences of storing their information in a public medium. What they want their friends to see, might not be seen by just their friends.

5.2 Removal of Confidential Information

We have already discussed the search patterns we used to discover sensitive information for this research. These and even more could be used by these services to check their own database systems for entries that could contain confidential information and then remove them or handle them in a way they deem appropriate. This surely has not been implemented in the past, but it is never too late to begin.

5.3 Automated Warnings

Another way to preserve user privacy and avoid storing links with personal information, is live-checking the long URL before it is shortened. If for example a user chooses to shorten a URL which contains his or her password information in plaintext, then the system could intelligently decline to offer the shortening service. This way the users can stop and think for a moment what they are trying to share and maybe even gain some insight into how badly certain systems they are using, are configured.

5.4 Expiration Dates

The short URLs could have an expiration date. As aforementioned, the rise in the use of such services came from social media. A lot of people choose to shorten URLs to share them over Twitter due to the limited amount of characters per message. But just like the way that the Internet popularity changes rapidly, the users tend to forget about those links after the trend has passed. That means that most of those links will have a high visitation rate close to their creation date, which will then start to drop as time progresses.

Our proposal would be to actually use the statistics and analytics that the APIs of the URL shortening services offer. One could call the APIs and check the creation date of a link, the amount of clicks it has gotten and be able to judge by those metrics if the links are still being used or not. If the fact that these links are no longer visited is evident, then they could be removed by the database and the URLs could actually be reused after a specific amount of time.

5.5 Deleting URLs

Finally, another solution would be to allow users to delete their shortened URLs. This type of services give the user the ability to remove URLs from their history or archive them, but the information still remains public and freely accessible to anyone who can find it.

6 Conclusions

During this research we learned how the URL shortening services operate and how the information publicized from users is being treated from them. These services operate in, a what seems, random manner when it comes to creating URLs but also with a certain level of sequentiality, which is obvious in the way the short URLs size increases.

6.1 User Privacy Implications

We witnessed the amount of available information to an attacker and discussed the consequences the use of these kind of services may have to the users' privacy and possible exposure of vulnerabilities in various systems which could be accessed in an unintentional and non standard manner.

We suggested possible solutions to these issues and made proposals to improve the user experience of the shortening services in ways that help make the users more aware of what they are actually using and how it works. And from the services' perspective we collocated our concerns and suggestions of how the privacy and intimacy of the users could be protected.

6.2 Sysem Security

Moreover we explained how feasible a much more intrusive approach is in collecting all this data (and the entirety of it if the time limit is large enough), how much time it would cost for someone to obtain this information and more importantly the size limitation that would occur in storing it.

If someone had the means and more importantly the intention of performing such a task with a malicious goal, they would be able to gather data and keep gathering and updating their database of these results and given enough time, be able to launch attacks to small or big systems, not only against individuals.

7 Future Work

In conclusion, we believe that our research could very well have a continuation as our limited amount of time did not allow us to research all the aspects that we initially hoped to. As already explained, it would not be very difficult for someone to gather the entire dataset of short URLs and perform an exhaustive research, given a large amount of time to collect the data and interact with it to discover all the sensitive information there is to be found if that was performed with some rather questionable media (e.g. botnets). If someone would want to do the same with a more honest approach, this research would take a lot longer, mostly because of the sheer amount of data there is to be found.

In addition we believe that there is a lot more information to be discovered in the HTML contents of these URLs. For obvious reasons such as lack of time and not less importantly legality, invasion of privacy and such we did not have the luxury of investigating this side of the research.

The main reason we believe there is a future for this research is that the dataset keeps growing constantly and unless there is a new method of sharing long URLs introduced soon, then the media which offer this type of service will not stop or change their use of it.

References

[1] Bots & Botnet: An Overview, SANS Institute, http://www.sans.org/reading-room/whitepapers/malicious/bots-botnet-overview-1299

[2] http://www.nirsoft.net/

[3] https://stat.ripe.net/

[4] https://stat.ripe.net/24.132.128.0%20#tabId=at-a-glance

[5] https://stat.ripe.net/58.183.128.0#tabId=at-a-glance

[6] https://stat.ripe.net/24.247.0.0%20#tabId=at-a-glance

[7] https://stat.ripe.net/158.109.64.164#tabId=at-a-glance

Appendix A Personal Contribution

Student	**Project**	**Report**
Alexandros	Literature Review, Data Research	All
Nikolaos	Code Development, Literature Review	All
Xavier	Code Development, Math Calculations	All

Table 3: Personal contribution in the project and report.

Appendix B Source Code