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 10, 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.

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 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 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 becomes public and easy to retrieve by anyone.

The rest of this chapter is dedicated to the detailed description of the problem we will be examining, presentation of previous work on this domain and mention of 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. Moreover, 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 take 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 and 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.

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 get to 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

2.1 Goo.gl

2.2 Bit.ly

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 Crawling

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 crawling 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 crawling 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 Mining

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

3.2.1 RegEx

3.2.2 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.

3.2.3 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.

3.2.4 MongoDB Queries

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 What can an attacker do?

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.

4.1.1 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 makes a lot more difficult for the URLS companies to detect the crawling 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 Atlas[3]. We selected various machines on different locations to get an estimation on the request times.

Server Location	**Minimum**	**Average**	**Maximum**	**Std Deviation**
Netherlands	7.254	7.316	7.439	0.035
Japan	257.859	265.058	269.539	4.787
USA (MI)	105.340	105.389	105.479	0.436
Spain	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 5.68 x 10^{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 crawling 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.

4.2 What have we found?

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 on 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.

- Google maps location and timestamps sharing: Some applications use the URLS 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 URLS can offer information on when a link was created, this can lead to a violation of a person intimacy, 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. We retrieved 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. This constitutes a major vulnerability that can be directly exploited by an attacker without any further effort.

- HTML crawling: 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 crawling 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.

****** I haven't added Xavi's references for the above from [1]-[7]. They are in the file he sent till we figure out if we will add them as footnotes or put them in the bibliography. ***********

4.3 User Privacy Implications

4.4 Sysem Security

4.5 Stats

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

7 Future Work

8 References

Wikipedia Definition:
$http://en.wikipedia.org/wiki/URL_shortening$

The case of URL Shorteners: How safe are they?
$https://www.stopthehacker.com/2010/02/19/analyzing-url-shorteners/$

What to consider before shortening URLs?
$http://www.polyu.edu.hk/ags/Newsletter/news1311/article3/index3.php$

URL shortening: Yet another security risk.
$http://www.techrepublic.com/blog/it-security/url-shortening-yet-another-security-risk/$

How to spy on campaigns of competitors who use URL shorteners.
$http://www.zdnet.com/article/how-to-spy-on-campaigns-of-competitors-who-use-url-shorteners/$

Stranger Danger: Exploring the Ecosystem of Ad-Based URL Shortening Services.

$https://lirias.kuleuven.be/bitstream/123456789/440951/1/strangerdanger_{w}ww2014.pdf$

Short Links under Attack: Geographical Analysis of Spam in a URL Shortener Network.
$http://kti.tugraz.at/staff/markus/documents/2012_{H}T2012_{S}hort_{l}inks.pdf$

Two Years of Short URLs Internet Measurement: Security Threats and Countermeasures.
$https://www.cs.ucsb.edu/\ chris/research/doc/www13_{s}horturls.pdf$

Security and Privacy Implications of URL Shortening Services.
$http://w2spconf.com/2011/papers/urlShortening.pdf$

we.b: The web of short URLs.
$http://www.csd.uoc.gr/\ hy558/papers/web.pdf$

9 Appendix

9.1 Personal contribution

9.2 Codes