Transparent Encryption for Cloud-based Services

Gergő Ládi
Department of Networked Systems and Services
Budapest University of Technology and Economics
Budapest, Hungary
me@gergoladi.me

***Abstract* — Transparent encryption is a method that involves encrypting data locally, on the user's computer, just before it is sent to cloud services to be stored, then decrypting said data later, straight after it is retrieved from the cloud service. All this takes place without having to alter the client application or the remote service (hence transparent). Applying this method ensures that even if the user's account or the provider itself is compromised, the attackers can only retrieve encrypted data that is useless without the encryption keys. This paper illustrates the design of a system that is capable of performing transparent encryption for various cloud-based services.**

Keywords – transparent encryption; cloud; security; DNS spoofing; tampering proxy; format preserving encryption;

I. Introduction

Over the course of the past five years, cloud-based services offering file storage, note and calendar management could be seen gaining ground over traditional self-hosted or serverless solutions, not just in the enterprise sector, but among home users as well [1]. According to a 2016 survey [2], the average consumer uses three different cloud services actively, at least one of which he uses daily. While these services offer many advantages to the end users, they present unique risks that are often overlooked:

- Users may lose their files if they don't use the service for a while and their account is deleted, or if the provider ceases operations without advance notification (as it was the case with MegaUpload[1] in 2011).
- If the provider is breached, anything that is stored by the users, including potentially sensitive documents or trade secrets, may be accessed by unauthorized individuals, possibly even published on the internet.
- If any website is breached where the user had an account, data stored at a cloud provider may be in danger if the user used the same password for both services, even if the cloud provider itself is otherwise reasonably secure. As of 20 April 2017, a total of 2,697,631,690 users had their records leaked from 210 different websites according to HaveIBeenPwned[2].
- The provider (its employees) may be able to access anything stored, all without the users' consent or knowledge. Even if the data is not directly accessed, the provider may still run data mining algorithms on it to build user profiles to be sold or used for marketing purposes.
- Finally, the provider may be forced to, or may decide to hand over user data to nation states or local authorities. This could put the users' lives at risk in countries where having opposing views to the party in power is enough to be threatened.

The first risk can be eliminated by having backup copies of everything that is stored at cloud providers, the third one by having different passwords for each and every service, and all the others by encrypting everything that is stored in the cloud. While some providers claim that they store everything encrypted, this does not help against risk no. three, and if the provider itself is assumed not to be trusted, it does not eliminate risks two, four, and five either. However, employing transparent encryption would solve all of the remaining issues as it could make sure that unencrypted data never leaves the user's computer.

While some vendors offer solutions (such as BoxCryptor) to secure certain cloud services, these solutions are limited to work with a given set of services, and are closed source. Being closed source means that it is not possible to audit the design or the implementation for security issues, and it also prevents developers from adapting it to similar services. In addition, existing solutions are usually not fully transparent in that they require the user to change the way he uses or accesses the cloud service. The solution proposed in this paper does not suffer from these limitations.

II. Cloud Application Message Sequences

In order to design a transparent encryption layer, one has to understand how cloud-based applications communicate with their servers. After having inspected six of the most well-known applications, I concluded that be it desktop, mobile, or browser-based, they all follow a common pattern. This pattern is depicted on Figure 1. , and is as follows:

[1] A popular file sharing and file storage service of the era that was shut down by the US Department of Justice for alleged willful copyright infringement.

[2] A website that collects dumps of database leaks from breaches and lets you check if you were affected by any of these.

1) When started, the application retrieves the hostname of the server where the remote service can be accessed. The hostname is usually stored in configuration files, but it might also be hard coded in the client executable. Then, a query is made to the DNS (Domain Name System) servers to resolve the server name to an IP (Internet Protocol) address.

2) A name server resolves the requested hostname and responds to the client.

3) The application initiates a TCP (Transmission Control Protocol) connection to the IP address. If the connection is successfully established, it attempts to secure the communication channel using TLS (Transport Layer Security).

4) Now that there is a secure channel, authentication proceeds. If successful, the user may read or modify data that is stored online. This is usually done via REST[3]APIs (Application Programming Interface) over HTTP (Hypertext Transfer Protocol).

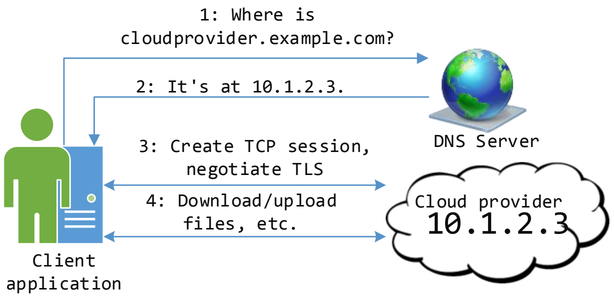

Figure 1. A typical cloud application message sequence (the IP address and the hostname are fictional)

It is not surprising to see REST APIs being used, since web-based protocols are widely supported in all environments, making it possible to use the same API for desktop, mobile, and browser-based versions of applications as well.

III. Designing Transparent Encryption

In order to perform encryption and decryption on user data, we need to divert the flow of data between the application and the cloud provider in a way that all data from the client to the server, as well as all data from the server to the client must pass through the transparent encryption layer. Neither the client nor the server should notice that they are not talking directly to each other anymore. It is also required that messages flowing through this layer be unprotected by TLS (or any other kind of encryption) so that the relevant messages may be identified and their contents may be changed in transit. This, in effect, describes a MitM (man-in-the-middle) attack.

A. *Diverting Traffic*

Since all cloud services rely on DNS, the diversion of traffic is most easily achieved by setting up a local DNS server that:

- resolves the provider's hostname to a local IP address (where the transparent encryption service is running) when asked by external applications,
- resolves provider's hostname to the actual IP address when requested by the transparent encryption service, and
- resolves everything else to their actual IP addresses, regardless of who the requestor is.

A possible method of differentiating between requests coming from the proxy service and external applications is using the reserved *.local* top-level domain [3] for internal requests in a way that if the original hostname was *a.example.com*, we query for *a.example.com._nospoof.local* internally. The local DNS server should be configured to resolve addresses according to these requirements.

B. *Handling Connections*

Once the DNS server is set up, requests to the cloud provider will be arriving at the local computer instead. To handle these, we need to design and implement a service that listens on TCP port 443 (the port of HTTPS – Secure HTTP)[4].

The service must be able to negotiate a secure (TLS) connection with the client, and in order to do so, a security certificate[5] is needed. These certificates aim to protect against exactly the same kind of MitM attack that we are performing, so further effort is needed to make this possible. Whilst some applications do not actually check the validity of certificates, but simply require their existence, this is bad security design and should not be relied upon. Since most applications delegate certificate validation to the operating system (or the browser, if running a browser application), it is possible to generate certificates that will be accepted as valid in most cases. First, we need to create a root CA[6], for example by using the *openssl* or *makecert* utilities. Then, we need to add the root CA's certificate to the list of trusted CAs. This ensures that certificates issued by this CA will be accepted as valid. Finally, we can use the root CA's certificate to issue and sign certificates for any domain, including that of the cloud service provider.

After a connection was established between the client and the proxy service, we also need to establish a connection to the actual cloud provider, then secure the connection using TLS. For this, we don't need a certificate, however, extensive care should be taken to validate the provider's certificate, otherwise we are opening ourselves up to MitM attacks by other parties.

---

[3] Representational State Transfer (REST): A kind of API that relies on HTTP as the layer 7 protocol, using HTTP verbs to indicate the action to be carried out (query, creation, modification, deletion), URLs to specify the resource to be manipulated, and HTTP's status codes to signal success/failure.

[4] In case of services that use a different protocol and/or a different port, this should be adjusted accordingly.

[5] Cryptographically verifiable evidence proving that a peer is indeed the one he is claiming to be.

[6] Certificate Authority (CA): an entity that can issue certificates

At this point, we now have the channels established and secured, and are just missing the message manipulation logic.

C. Inspecting Traffic

Having reached this point, it is possible to read the client's requests (unencrypted) from the client-to-proxy connection, interpret them, change message contents if desired, and then write the messages to the proxy-to-provider pipe. Processing responses from the provider is analogous (with the pipes swapped). Since the protocol used above TLS is HTTP, it is suggested that HTTP libraries be used to parse messages. This eliminates the need to manually decompress messages, process and interpret headers and convert between character sets, saving the programmer from a series of potentially dangerous pitfalls. Furthermore, the data types used within the HTTP requests are JSON[7] or XML[8]/SOAP[9], with several serialization (deserialization) libraries available for both.

In order to discover the message types and data structures used by a particular service, one can first design the proxy in a way that it does nothing but relay requests and responses unaltered, while also dumping messages to a file or database. Setting up the proxy and using the cloud application for a while should uncover most message types. Services often offer publicly available APIs to developers so that they can interface with the cloud service from 3rd party applications. It is recommended to check these APIs, as even if the endpoints are different, the message types and the data structures might be the same or similar.

D. Altering Traffic

Once we understand the message types and data structures, we need to decide what should be protected. Typical candidates are file contents, text fields, dates, phone numbers, and e-mail addresses. After the relevant fields are identified, one can make a list of requests that contain these, then create filters based on the API endpoints or message signatures. The filters should be chained together to inspect each request and response that passes through, deciding whether the current filter should alter the current message before passing it on to the other end of the pipe. In this case, altering messages means encrypting or decrypting certain fields of the data structure.

Putting all of the above together results in a system (Fig. 2) that can transparently encrypt data that is being sent to a cloud service provider, then decrypt it on the way back.

IV. Format Preserving Encryption

Cloud providers typically perform format and range validation on anything that is submitted to the service. For this reason, the naïve idea of encrypting fields with a usual stream cipher, then sending resulting ciphertext to the service will not work, since the raw binary data will not pass validation checks. While this could sometimes be worked around by applying Base64 encoding [10] to the binary data, this unnecessarily increases the length of the output, and APIs often impose maximum length restrictions. This is where format preserving encryption algorithms (FPEs) are useful.

An encryption algorithm $\mathcal{F}$ is said to be format preserving if the domain and the range (the $\mathcal{M}$ message space) are the same (with the exception that the algorithm also takes a key parameter $\mathcal{K}$) [4].

$$\mathcal{F}: \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{M} \tag{1}$$

Using such algorithms, we can encrypt data in a way that the ciphertext passes format validations from simple length checks to more complex range or integrity checks.

The first format preserving algorithms with provable security were described by Black and Rogaway [5], who proposed three possible approaches:

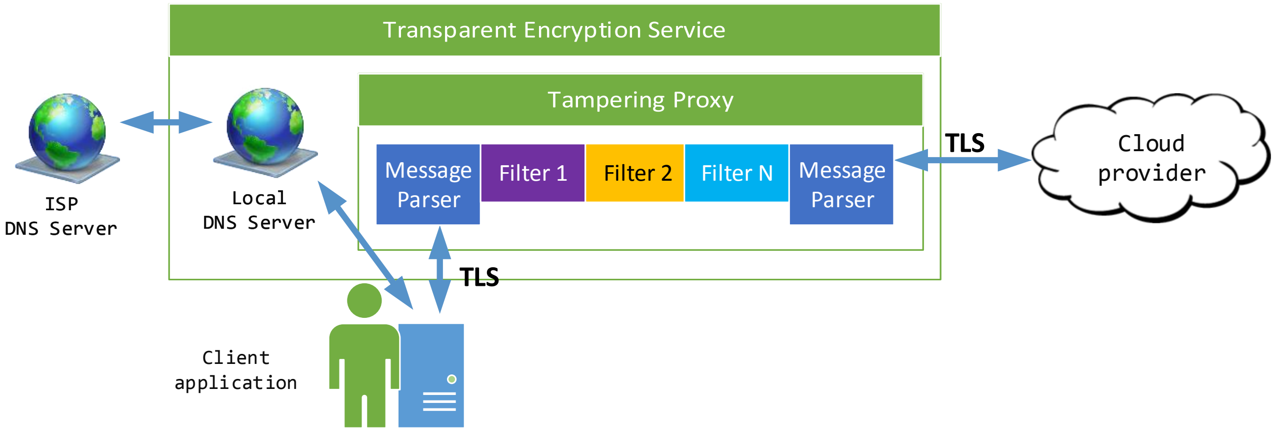

Figure 2. The architectural diagram of the transparent encryption system

[7] JavaScript Object Notation (JSON): a notation that uses JavaScript-like syntax to describe data structures. It is often used in applications that have a web-based front-end since it is easy to work with JSON from JavaScript.

[8] Extensible Markup Language (XML): a markup language with well-defined rules for encoding documents and messages (data structures).

[9] Simple Object Access Protocol (SOAP): a method for exchanging data structures and invoking remote procedures. Uses XML for messaging.

[10] A two-way transformation that transforms its input in a way that the output contains only the non-capital and capital letters of the alphabet, the ten numbers, and two other characters: '=' and '/'.

- a prefix cipher-based construction that is only effective for small domains as we have to store a lookup table of a size that of the domain,
- a construction based on cycle walking, which does not use a lookup table, but is a recursive function that may take several cycles to complete (as such, its runtime is unpredictable), and
- a construction based on Feistel networks[11] that does not need a lookup table, but may need cycle walking (however, the number of rounds can be limited by tweaking the parameters).

Based on the above, we can see that the Feistel network-based algorithms are the best for general use. Multiple implementations exist, typically differing in the number of rounds, the maximum supported range of values, and whether the algorithm takes a tweak[12] or not. These algorithms, however, are only capable of transforming integers of a given range to other integers in the same range. To overcome this limitation, ranking functions can be used to map the elements of the message space to integers.

For example, for simple text fields, we can use a function that maps each letter to its (1-based) position in the alphabet, then encrypt this rank value using the cardinality of the alphabet as the maximum value parameter to the FPE algorithm. Date-and-time type fields can be expressed as the number of seconds that passed since a given reference date. This number, together with a desired maximum offset value can be used as parameters to an FPE algorithm [6].

V. Further Considerations

This section is meant to briefly introduce how the system could possibly be improved in the future, as well as highlight features that might make it unfavorable to use in certain cases.

A. *Possible Threat: Ever-changing APIs*

For the system to work properly, the filters have to be able to recognize the message types and data structures. If the cloud service provider keeps making frequent changes to the API, the maintainers of the transparent encryption layer will also have to make frequent changes to the code of the filters. In the meantime, user data might be inaccessible, and unencrypted information might leak to the provider.

B. *Possible Threat: New Security Measures*

New security measures such as certificate pinning[13] or HSTS[14] might make it impossible to perform the MitM attack. Disabling these might require the application to be patched, at which point the method is no longer transparent.

C. *Supporting Multiple Services*

The system could be improved to support multiple cloud services side-by-side, by modularizing the proxy further, adding a dispatcher that routes incoming requests to the appropriate module based on the *Host* header received during TLS negotiation.

D. *User Key Management*

With the current design, if the user intends to use the cloud service on multiple devices, he has to copy the key file to each device manually. By employing a password-based key derivation function, it should be possible to generate the same key separately on each device, without having to transport files.

Conclusion

Cloud-based services are popular and will stay popular in the near future. They bring with themselves several risks from a security standpoint that are often underestimated. The aim of this work was to elaborate a method to increase the security of cloud-based applications, even in cases where the cloud service provider cannot be trusted at all. The proposed solution relies on hijacking DNS queries and performing MitM attacks against certain SSL/TLS sessions, then analyzing and selectively encrypting/decrypting message contents.

Related Works

A working proof of concept having the previously discussed architecture and properties was implemented in C# for Google Calendar and the note-taking service Evernote as part of a different project.

Acknowledgment

I would like to express my thanks to my advisor Dr. Levente Buttyán and the CrySyS laboratory for the useful advice and support.

References

[1] S. Nag, L. Lam, Y. Dharmasthira et al, “Forecast: public cloud services, worldwide, 2014-2020, 2Q16 update,” Gartner, G00310051, 2016.

[2] K. Weins, "Cloud computing trends: 2016 state of the cloud survey," (online), http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey, 2016

[3] S. Cheshire, M. Krochmal, Apple Inc., “Multicast DNS,” (online), https://tools.ietf.org/html/rfc6762, Internet Engineering Task Force (IETF) RFC 6762, 2013, p. 5.

[4] J. Black, P. Rogaway, “Ciphers with arbitrary finite domains,” RSA-CT, 2002, p. 114.

[5] M. Bellare, T. Ristenpart, P. Rogaway, T. Stegers, “Format-Preserving Encryption,” Cryptology ePrint Archive: Report 2009/251 (online: https://eprint.iacr.org/2009/251), 2009

[6] Z. Liu, C. Jia, J. Li, X. Cheng, “Format-preserving encryption for DateTime,” IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS), 2010

[11] A Feistel network is a symmetric iterated cipher construction that uses an internal round function. The symmetry makes it possible to use the same construction for both encryption and decryption by changing the order of the parameters.

[12] An extra parameter used during encryption that helps ensure that even if the key and the plaintext are the same, the ciphertext will be different. This is similar to how Initialization Vectors (IV) work for other algorithms.

[13] A security check that, in addition to requiring a valid certificate, also requires that the certificate have a specific serial number or be issued by one of the certificate authorities on a list.

[14] Hypertext Strict Transport Security (HSTS): a security mechanism that can be used to enforce HTTPS and make certificate checks stricter.