RESTful Security

Dan Forsberg
Nokia Research Center
Helsinki, Finland
dan.forsberg@nokia.com, dforsber@gmail.com

***Abstract*— We take a look into the REST architectural style of making scalable web applications and find out the critical requirements that mismatch with the current web security and privacy architecture. One of the core challenges is the inability of the web security model to scale up with caching when millions of users share confidential data inside communities. Our contribution includes a new solution for achieving RESTful security for web architecture without secure URLs. The solution scales up the performance of web services that require confidentiality protection and relaxes the security requirements for data storage networks by separating the access control decision from the data request.**

REST, web security, TLS, web caching, user privacy, key hierarchy, key derivation

I. INTRODUCTION

More applications are being developed for the web instead of as native applications for an operating system like Windows. Social networking is one common phenomenon for these applications and allows people to register, create own profiles, tune their application preferences, and invite friends to join communities. The users upload photos and other personal data and share information about their life, like how they think, live, consume, and connect with different people. This brings up security issues like user's privacy, data confidentiality, identity verification (authentication), and access authorization for handling all this personal data. Who is able or allowed to access the data? What is considered to be private and public and how it is followed? It is crucial that scalable security is taken into account and built into the architecture of applications and services like these in the open internet with billions of users.

The current methods for implementing security include user authentication [1] and Transport Layer Security (TLS) [2] for protecting sessions over the Internet. Certificates are used to authenticate the web site URLs for the clients, but the scheme wrongly relies on human understanding of the links and certificates and thus phishing attacks have emerged. Web cookies [3] are used to e.g. transfer session information and to carry authorization information in the HTTP requests during the session lifetime. The personal data of the users is usually stored and in many cases transferred unencrypted. Furthermore, users have weak or no control over the data that is once transferred to the services. If a malicious user is able to access another person's (victim) picture or video and put it into the Internet the victim has small or no chances to delete the content once it has spread around. The only defense may be a secret URL, transferred in plain text over the network, which may not be good enough in some cases as the URLs can be sniffed by others. Actually, the users may copy and publish the links by themselves.

There are many ways to design and implement web applications. Many applications are implemented with the model of Remote Procedure Calls (RPC over HTTP) that are executed on the server side and thus increases the load. This does not help service providers to easily scale up their services for a higher number of clients, which is a crucial requirement for today's web applications. One of the answers to this problem is Roy Fielding's Representational State Transfer (REST) [4..7] architectural style for web applications and has become an important set of requirements for modern web application developers and designers (e.g. REST APIs). REST style brings clarity on how to build highly scalable web applications. For example, it allows servers to be more stateless and utilizes the benefits of caching with more static Uniform Resource Identifiers (URI) [8]. In REST style applications the URIs can be referenced after the session as well in contrast to many web applications, where the dynamic URIs are used and their relevance is low after the session ends. Even REST is described as an architectural style, it implies multiple requirements for web applications. It efficiently utilizes the HTTP protocol (version 1.1 [9]) methods to handle data and requests in contrast to web applications that use single GET method to invoke remote scripts with arguments to modify and read data.

There is a gap between the REST architecture and the current security features of today's web. The security architecture does not naturally align with the REST architecture in the sense that secure sessions create session specific keys but more static data that can be stored in web caches can not be confidentiality protected and fetched from the caches at the same time. This heavily reduces the scalability of the REST architectural style for applications and services that require access control to the data and for this reason provide the data through e.g. TLS [2] tunnels or require HTTP authorization. In this paper we identify and address some of the challenges that rise up from this gap.

In section 2 we describe the REST architectural design style and web caching in more details. In section 3 we describe some possible solutions to overcome the limitations of the current security architecture that also support the REST architectural style for web applications. Furthermore, we analyze our solution in section 4 and list some issues for further study. We conclude this paper in section 5.

II. REST ARCHITECTURAL DESIGN STYLE ALIGNS WITH WEB CACHING

HTTP version 1.1 has four main methods for client requests, namely GET, PUT, POST, and DELETE (there are also other methods like HEAD, CONNECT, and TRACE, which we do not address in this paper). The REST handles all data as URIs and the HTTP methods are applied to them. To make this more general, the HTTP GET can be seen as similar to read data, PUT similar to create/replace data, POST similar to append to/create data, and DELETE similar to remove data. GET (also HEAD) is a safe read method, which does not alter the data, but all the other methods update the data in some ways and can be thought as write methods (i.e. append, replace, or remove).

Web content caching [10] with the URIs assigned to the data items is an important part of RESTful thinking and applies to the HTTP GET method. Data that needs to be presented to the user via the browser is fetched with the HTTP GET method from the web servers. Between the client and the server there can be web proxies and web caches that may already contain the requested URI presented in the GET request. Caches reduce bandwidth usage and especially the server load, and shows as smaller lag to the user. On the other hand the freshness of the fetched data needs to be known.

There are multiple web caching models. User agent caches are implemented in the web browsers in the clients themselves and are user specific. Proxy caches (also known as forward proxy caches) are most known to normal users as they require configuration of the browser (i.e. proxy settings). Interception proxy caches or transparent caches are variants that do not require setting up the clients. On the other hand gateway caches, reverse proxy caches, surrogate caches, or web accelerators are closer to or inside the server site and not visible to the clients either. There are protocols to manage the contents of the web caches in a distributed manner, such as Internet Cache Protocol (ICP) [10, 11] and Hypertext Caching Protocol (HTCP) [12]. Further on, the web caches can work together to implement Content Delivery (or Distribution) Networks (CDN). These become very important when the scalability of video on demand services like YouTube (www.youtube.com) etc. is considered.

HTTP protocol includes mechanisms to control caching. Freshness ("cache lifetime") allows the cache to provide the response to the client without re-checking it on the origin server. Validation is used in the cache to check from the origin server whether the expired cache entry is still valid. Then, an important feature for the RESTful architectural model is the way how cache entries may become invalidated. Invalidation happens usually as a side effect when HTTP PUT/ POST/ DELETE request is applied for the respective cached URI [9]. Since these requests modify the respective URI the cache can not provide the cached version of the URI back to the client but let the origin server handle the write operation and provide the response (note that there may be other web caches on the routing path that are not traversed, especially user agent and proxy caches). On the other hand if the HTTP GET method is designed to be used as an RPC method to call a script in the server for writing data, the cache may think it has a valid response in the cache already for the URI and return an old response. This may be ok for the application or service logic. REST architectural style of implementing web applications gives a good guidance for the designer and developers. It is about understanding the nature of the web and not misusing it. It also discourages the usage of scripting for all user session specific data handling as the content based on the results from scripts are not generally cached.

The REST style encourages having a separate URI for each data item, like a single photo or entry in a database. One of the reasons is that different data items can be cached separately, e.g. a user's image in the cache does not expire even if the user changes the profile data information in the web application database. This encourages developers to apply HTTP PUT/ POST/ DELETE to a most accurate URI in question. In contrast one might design the web application in such a way that all PUT/ POST/ DELETE queries go to the same root URI but with different arguments for the script. This may flush the cache [9] as the data is updated with these methods and the current cached entry may become invalid. Using scripts also makes effective caching hard for all entries addressed with the root URI if for example the `mod_cache` [13] is used with Apache web server. Take this search query URI as an example:

(1) http://mypics.com/?cmd=create&cat=music&sub=rock&title=acdc

and compare it with the following examples:

(2) http://mypics.com/music/rock/?title=acdc
(3) http://mypics.com/music/rock/acdc

We see that the first example, if used with PUT/ POST, may disable cached copies of all entries for the mypics.com (write operation on that URL updating the content), whilst the second only for the rock subcategory in the music category. The last example row is the simplest and follows the RESTful design, e.g. if used with PUT. All GET, PUT, POST, and DELETE can be invoked with the same URI and the web application knows what to do with it.

A. *Web security and caching*

There are message based end-to-end security mechanisms, like Secure / Multipurpose Internet Mail Extensions (S/MIME) [14], Cryptographic Message Syntax (CMS) [15], and Pretty Good Privacy (PGP) email protection program (www.pgp.com) based on public keys originally developed by Philip Zimmermann back in 1991. S/MIME is a public key based standard for encrypting and signing emails, it adds the security extensions into the MIME. CMS is a more general specification for message level authentication, signing, and encryption. CMS supports shared secrets in addition to public key based solutions. These are application level security mechanisms and not generally implemented for securing web content.

With TLS the secure sessions are user specific and keys are generated on the fly. The content that is encrypted again and again when pushed to the secure tunnel. The data can not be cached as the web caches can not access the data inside the secure tunnel. On the other hand the client accessing the content gets the data and can further copy it locally or

distribute it further (even the decryption keys themselves). The former problem is about content protection and access control, latter problem about user control and platform security and relates to DRM, which is out of the scope of this paper. Thus, in this paper we concentrate on the former problem, namely caching content that requires access control and confidentiality protection.

III. DIRECTORY SPECIFIC SYMMETRIC CONTENT ENCRYPTION KEYS

Let's say we build a "mypics.com" web application in a RESTful style and make the URIs in a way that the pictures can be easily cached for scalability reasons. But with the current web technology if they are cached, anybody can get the pictures if they know the correct (secret) URI even if they were not authenticated users. So, we have a problem. If we apply secure sessions with TLS and transfer all the pictures inside the secure tunnel, caching is effectively disabled as the caches along the path can not intercept the pictures. Also, the pictures are encrypted multiple times for each client accessing them (redundant encryptions). But on the other hand the pictures are safe and user sessions authenticated. Another possible improvement solution is to disable general caching and apply application specific caching near the server where the secure session ends. However, this is not a nice solution as it requires application platform specific caching and does not utilize the benefits of proxy caches. And still, it also requires that the server encrypts the pictures as they go through the tunnel for each session separately (redundant encryptions). It also puts security requirements on the data in the servers behind access control.

In user communities photos are generally shared among trusted people only. This requires access control to the photos and user authentication. Also, commercial sites that require users to pay for the content want to restrict the content to the customers that pay for it but at the same time want to make their service architecture as scalable as possible to allow higher growth of customer base. There is a mismatch between these two targets when web caching is considered. I.e. web caching is not possible for the encrypted content.

Our solution sketch to this problem is simple. We create a content protection key, optionally bind the lifetime of that key with the cache lifetime of the URI, and provide the key to those clients who are authorized to access the ciphered content. All pictures (or e.g. videos with progressive download) are encrypted with the content protection key and can thus be stored outside the service provisioning pool. We require secure user authentication and content access authorization decisions (e.g. through TLS tunnels or with HTTP authentication and authorization mechanisms). The actual data is then accessed without secure HTTP and thus served for the clients directly outside secure TLS tunnel or HTTP authorization headers even from any available cache that may have the encrypted data stored. In this model the secure HTTP session acts as a control channel where the data protection keys are provided for the clients after proper authentication and authorization. We support the REST architectural style and allow all clients to access the encrypted data content URIs without access control. Thus, the data is not usable for the clients if they do not have the keys to decrypt the content. Note that the end result of this model is similar to current web security model with TLS, except that the (a) servers do not do redundant encryptions and that the (b) caching of the data under access control is possible without any needs to make the URLs secret.

We use key hierarchies together with directory hierarchies for supporting REST architectural style. In this model we have a root key root-K for the root directory ("http://www.domain.com/") and derive next level keys along with the directory structure. For example:

http://www.domain.com/ : root-K
…/pictures/ : pictures-K = H(Root-K, "pictures")
…/pictures/john/ : john-K= H(Pictures-K, "john")
…/pictures/john/23.jpg : 23.jpg-K = H(John-K, "23.jpg")
…/pictures/mary/ : mary-K = H(pictures-K, "mary")
…/pictures/mary/face.jpg : face.jpg-K = H(mary-K, "face.jpg")

Here the function H is a one-way key derivation function (KDF) used for getting next level keys. The keys are bound to the directory and file names. This allows access control based on every single file or set of directories below a root directory. Each client knows how to create the next level key from the root key. Thus, John could set the policy to allow Mary to get the key John-K, but set the policy to allow Jane to see only the picture 23.jpg and thus get the key 23-K only. Note that the names of the keys are in the scope of the directory namespace, i.e. face.jpg-K key is not unique name until it is bound with the directory of /pictures/mary/.

A. *Extending the key hierarchy*

We initially wanted to align the web security architecture with the REST architectural style without loosing the existing content protection and access control features. In our model the content is protected and the keys are provided only for those clients or users who have authorization to access the content. Our model also scales well to different levels of security policies where the subdirectories and files are protected with separate keys based on a key hierarchy. However, the disadvantage of this model is that once a client gets a key for a directory all the subdirectories are also accessible for the user.

The model could be extended to break the key hierarchy in these cases and create a new and independent root key for the subdirectory that requires access control separation from the parent directory's access control. This would also require the client to understand that now the directory key can not be used to create subdirectory key for this particular subdirectory. The immediate analogy to the web server configuration would be to use the .htaccess file and extend it to support a mechanism, which describes the key hierarchy relation for that particular directory. For example the .htaccess file could say that the content protection key is NULL (no encryption), PARENT (use the parent directory key to derive the subdirectory key), or ROOT (start a new key hierarchy for this directory).

The downside of this extension is that the client does not know whether new key is needed or if the existing parent key can be used. Without this extension the client could always apply the parent key for directory hierarchies with assigned root keys. One way to implement this extension would be to

extend HTTP headers with information about the key hierarchy root, adding a key identifier of the respective key, or both.

IV. DISCUSSION

There are multiple advantages in our approach of content protection based on separate content protection keys delivered to clients via secure authenticated and authorized sessions. First and most important advantage is that this way the web security architecture could be better aligned with the REST architectural style. Doing this enables all the caching scalability advantages of the REST architectural design style.

Our model reduces server load considerably with content that requires confidentiality protection. When using a content protection key, there is no need to encrypt the same content again and again over a TLS session. The server needs to protect the content once per caching lifetime, which can be very long for content that does not change, e.g. picture and video files.

Clients can get the content from local or remote caches even without logging in into the web application once they have received the content protection keys. This also effectively enables offline use cases for web applications that require content protection and access authorization. On the other hand our model also allows content pre-distribution to clients and getting the access rights and content decryption key later on. This may have potential to improve the user experience on some services, where the content can be downloaded in the background and displayed for the user immediately when decryption keys are available (pre-fetching or parallel fetching while user is authenticating).

There are multiple implementation alternatives. Initially if a web application developer wants to use this model of encrypted content, the encryption, decryption, and key management could be done on the application layer (e.g. with Javascript and plugins). Another approach could be to standardize needed extensions with the HTTP protocol. There is lots of work to be done for further defining how this model would work in practice and actually verifying how different caching technologies behave. CMS could be used for the actual content encryption (and integrity protection).

V. CONCLUSION

We analyzed the web security architecture model within the scope of REST and concluded that the two of them are not well aligned. To overcome this mismatch, we sketched a solution, which consists of hierarchical content protection keys that share the lifetime of the cached context and are delivered to the clients through secure HTTP after proper user authentication. We analyzed this solution in high level and gave some topics for further study on this area. Our solution seems to give a promise of improved web application scalability in cases where access control and content protection needs to be applied for all content. This is a very important topic as there are multiple content distribution network providers and hundreds of millions of files that require access authorization. We also realized that it is not easy to find out how caching works in real life as there are many configuration options.

Our solution reduces the need to do HTTP authorization checks for the data as access is controlled with the decryption key. In effect our solution is analogical to secret URI usage schemes except that instead of adding the secret part into the URI, we use key to secure the content itself, which better fulfills the privacy and data confidentiality needs. Our solution could also be used e.g. with OAuth [16], where the Access Token is replaced with the content decryption key.

Getting the data to the client is only part of the solution of access control to the content. Users can copy the images and videos to other users, which essentially bypasses the access control of the web applications. This said, we note that our solution is aligned with the model of the current web security model but adjusts it to match better with web caching for data that requires confidentiality protection.

REFERENCES

[1] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Sink, E. and L. Stewart, HTTP Authentication: Basic and Digest Access Authentication, Internet RFC 2617, June 1999. URL: ftp://ftp.isi.edu/in-notes/rfc2617.txt

[2] T. Dierks, C. Allen, The TLS Protocol Version 1.0; Internet RFC 2246, January 1999, URL: http://tools.ietf.org/html/rfc2246

[3] "HTTP State Management Mechanism"; D. Kristol, L. Montulli, October 2000, Internet RFC 2965, URL: http://tools.ietf.org/html/rfc2965

[4] "Architectural Styles and the Design of Network-based Software Architectures", Roy Fielding, Ph.D. thesis. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[5] Ryan Tomayko, "How I explained REST to My Wife", Referenced 2008-11-18, URL: http://tomayko.com/articles/2004/12/12/rest-to-my-wife

[6] Stefan Tilkov, "A brief introduction to REST", Referenced 2008-11-18, URL: http://www.infoq.com/articles/rest-introduction

[7] Leonard Richardson, "RESTful Web Services", Referenced 2008-11-18, URL: http://www.amazon.com/RESTful-Web-Services-Leonard-Richardson/dp/0596529260

[8] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. Internet RFC 2396, Aug. 1998, URL: http://tools.ietf.org/html/rfc2396

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "HTTP 1.1 Protocol", Internet RFC 2616, URL: http://www.w3.org/Protocols/rfc2616/rfc2616.html)

[10] D. Wessels, K. Claffy, "Internet Cache Protocol (ICP), version 2"; Internet RFC 2186, Sept 1997, URL: http://tools.ietf.org/html/rfc2186

[11] D. Wessels, K. Claffy, "Application of Internet Cache Protocol (ICP), version 2", Sept 1997, Internet RFC 2187, URL: http://tools.ietf.org/html/rfc2187

[12] P. Vixie, D. Wessels, "Hyper Text Caching Protocol (HTCP/0.0)", January 2000, Internet RFC 2756, URL: http://tools.ietf.org/html/rfc2756

[13] Apache HTTP server mod_cache module documentation; Referenced 2008-11-18. URL: http://httpd.apache.org/docs/2.2/caching.html

[14] B. Ramsdell, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification", Internet RFC 3851, July 2004, URL: http://www.ietf.org/rfc/rfc3851.txt

[15] R. Housley, "Cryptographic Message Syntax (CMS)", Internet RFC 3852, July 2004, URL: http://www.ietf.org/rfc/rfc3852.txt

[16] Mark Atwood & al., "OAuth Core 1.0"; URL: http://oauth.net/core/1.0/, December 4, 2007