Security Metrics in Web Services

Shabnam Kumari[1], Deepak[2], Sunita Kumari[3]

[1]A.P., Department of CSE, Sat Kabir Institute of Technology & Management, Bahadurgarh, Haryana, India
[2]Mtech scholar, Department of CSE, Sat Kabir Institute of Technology & Management, Bahadurgarh, Haryana, India
[3]A.P., Department of CSE, , G.B.Pant College of Engineering, Okhla, New Delhi, India

Abstract*-Metrics are tools designed to facilitate decision-making and improve performance and accountability through collection, analysis, and reporting of relevant performance-related data. Security metrics are valuable for measuring and comparing the amount of security provided by different systems and configurations. 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.*

Keywords-Metrics, data confidentiality, web services.

I. SECURITY OVERVIEW

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 and Transport Layer Security (TLS) 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 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) 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).

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) 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 tunnels or require HTTP authorization.

II. WEB SECURITY AND 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 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) and Hypertext Caching Protocol (HTCP). 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. 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 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 is used with Apache web server.

Take this search query URI as an example:
http://mypics.com/?cmd=create&cat=music&sub=rock&title=acdc
and compare it with the following examples:
http://mypics.com/music/rock/?title=acdc
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.

III. CHOOSING RIGHT PROTOCOL

Industry standard authentication protocols help reduce the effort of securing your API. Custom security protocols can be used, but only under very specific circumstances. Here is a brief overview of the benefits and drawbacks of the top protocols.

3.1. Basic API Authentication w/ TLS

Basic API authentication is the easiest of the three to implement, because the majority of the time, it can be implemented without additional libraries. Everything needed to implement basic authentication is usually included in your standard framework or language library. The problem with basic authentication is that it is, well "basic", and it offers the lowest security options of the common protocols. There are no advanced options for using this protocol, so you are just sending a username and password that is Base64 encoded. Basic authentication should never be used without TLS (formerly known as SSL) encryption because the username and password combination can be easily decoded otherwise.

3.2. OAuth1.0a

OAuth 1.0a is the most secure of the three common protocols. OAuth1 is a widely-used, tested, secure, signature-based protocol. The protocol uses a cryptographic signature, (usually HMAC-SHA1) value that combines the token secret, nonce, and other request based information. The great advantage of OAuth 1 is you never directly pass the token secret across the wire, which completely eliminates the possibility of anyone seeing a password in transit. This is the only of the three protocols that can be safely used without SSL (although you should still use SSL if the data transferred is sensitive). However, this level of security comes with a price: generating and validating signatures can be a complex process. You have to use specific hashing algorithms with a strict set of steps. However, this complexity isn't often an issue anymore as every major programming language has a library to handle this for you.

3.3. OAuth2

OAuth2 sounds like an evolution of OAuth1, but in reality it is a completely different take on authentication that attempts to reduce complexity. OAuth2's current specification removes signatures, so you no longer need to use cryptographic algorithms to create, generate, and validate signatures. All the encryption is now handled by TLS, which is required. There are not as many OAuth2 libraries as there are OAuth1a libraries, so leveraging this protocol for REST API security may be more challenging.

Last year, the lead author and editor of the OAuth2 standard resigned, with this informative post.. Because of this instability in the spec committee and because OAuth2's default settings are less secure than OAuth1 (no digital signature means you can't verify if contents have been tampered with before or after transit), we recommend OAuth1 over OAuth2 for sensitive data applications. OAuth2 could make sense for less sensitive environments, like some social networks.

IV. CUSTOM PROTOCOLS

Custom API authentication protocols should be avoided unless you really, really know what you are doing and fully understand all the intricacies of cryptographic digital signatures. Most organizations don't have this expertise, so we recommend OAuth1.0a as a solid alternative.

Even if you are willing to take this potentially perilous road, there is another reason to avoid it: because it is custom, no one other than you will be able to use it easily. Only use custom authentication protocols if you are willing to support client libraries you can give to your REST API callers (Java, Ruby, PHP, Python, etc) so your users can use these protocols with little or no effort. Otherwise the API will be ignored.

V. CONCLUSION

As Web services are still relatively new in terms of their practical implementation, web architects and developers need to be careful in how they deploy Web services. In addition to the protective measures discussed in this document, standard recommendations for the security of web applications should also be followed.

Some best practices are:

1. Harden underlying servers according to security guidelines.
2. Apply the latest security patches to all system components.
3. Ensure that strict validation is applied to all input.
4. Ensure proper authentication and authorisation is enforced to restrict privileges and access rights to only valid personnel.

In addition, when firewalls do not provide adequate security when it comes to the deployment of Web services, a WS-Security or XML-aware gateway should be considered.

REFERENCES

[1] Scott Berinato, A Few Good Information Security Metrics, CSO Magazine, July 01, 2005, http://www.csoonline.com/article/220462/A_Few_Good_Information_Security_Metrics?contentId=220462&slug=&

[2] The CIS Security Metrics Service, The Center for Internet Security (CIS), July 1, 2008, http://securitymetrics.org/content/attach/Metricon3.0/metricon3-kreitner%20handout.pdf

[3] Maxwell Dondo, A Fuzzy Risk Calculations Approach for a Network Vulnerability Ranking System, Technical Memorandum 2007-090, Defence R&D Canada – Ottawa, May 2007, http://www.ottawa.drdcrddc.gc.ca/docs/e/TEO-TM-2007-090.pdf

[4] Colin Wong and Daniel Grzelak, "A Web Services Security Testing Framework", SIFT SPECIAL PUBLICATION, Information security services, Version 1.

[5] john Steven and Gunnar Peterson,"A Metrics Framework to Drive Application Security Improvement", IEEE Security & Privacy, vol. 1, no. 4, 2003, pp. 88–91. H. F. Tipton and M. Krause, Information Security Management Handbook, CRC Press, 2004.

[6] effreyR. Williams and George F. Jelen, "A Practical Approach to Measuring Assurance",Document Number ATR 97043, Arca Systems, Inc. , 23 April 1998

[7] Bachar Alrouh and Gheorghita Ghinea, "A Performance Evaluation of Security Mechanisms for Web services", 2009 Fifth International Conference on Information Assurance and Security