Report # I73-015R-2011
Date: 3/25/11

Guidelines for Implementation of REST

Enterprise Applications Division
of the
Systems and Network Analysis Center (SNAC)

Information Assurance Directorate

National Security Agency
9800 Savage Rd. Suite 6704
Ft. Meade, MD 20755-6704

Table of Contents

Contents

I. Introduction

Representational State Transfer (better known as REST) is a programming philosophy that was introduced by Roy T. Fielding in his doctoral dissertation at the University of California, Irvine, in 2000.[1] Perhaps the best description of REST is from Roy Fielding himself:

> "Representation State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."[2]

It is important to note that REST is not a standard, but rather an architectural / programming philosophy or paradigm. However, REST does promote the use of standards such as HTTP, URI, XML and JSON and formats such as GIF, MPEG, etc. Examples of REST usage in the commercial world are Twitter, iPhone apps, Google Maps, Salesforce integration, Programmable Web.com, and Amazon Web Services (AWS).

Although REST is not a standard, there are some standard practices that can be used to implement REST in a more secure manner. This paper will provide high level guidance for implementing REST in a secure manner and references to additional resources. The goal is to be able to take advantage of REST (utilizing HTTP/S as the implementation standard) while not compromising the security of enterprise systems.

II. Background

REST was first discussed in Roy Fielding's doctoral dissertation in 2000. Since then it has been gaining popularity and is being used in many different areas. The appeal of REST is its simplicity – it *generally* relies on the HTTP protocol and its verbs GET, PUT, POST and DELETE (although technically REST does not have to depend on HTTP). REST is stateless, is cacheable and relies on uniform named resources (Uniform Resource Identifier - URI) identifying unique resources. REST does not specify implementation details other than the use of the four HTTP verbs, the naming of resources using nouns (e.g. URI), and the interconnection of resources with URIs. The World Wide Web is a good example of REST at work. HTTP is used to access information which is identified with a URI. REST permits scalability and general interfaces (HTTP) that allow for simplified communication.

---

[1] Fielding, Roy T., "Architectural Styles and the Design of Network-based Software Architectures," http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000.

[2] Costello, Roger L., "REST (Representational State Transfer)," http://www.xfront.com/sld005.htm, slide 5.

The key to successful use of REST is the ability to understand the data set, identify the key resources, identify the relationships between these key resources, identify the operations allowed on resources, and the ability to address the data via URIs. There are no specialized tools required to implement REST.

A. REST - What is it?

As previously mentioned, Representational State Transfer (REST) is an architectural principle rather than a standard or a protocol. The basic tenets of REST are: simplify your operations, name every resource (with a URI), and utilize the HTTP commands GET, PUT, POST, DELETE according to how their use is outlined in the original HTTP RFC (RFC 2616[3]). REST is stateless, does not specify the implementation details, and the interconnection of resources is done via URIs.[4] REST can also utilize the HTTP HEAD command primarily for checking the existence of a resource or obtaining its metadata.

B. Advantages and Disadvantages

1. Some of the advantages of using REST include:

    - Every resource and interconnection of resources is uniquely identified and addressable with a URI [consistency advantage]
    - Only four HTTP commands are used (HTTP GET, PUT, POST, DELETE) [standards compliance advantage]
    - Data is not passed, but rather a link to the data (as well as metadata about the referenced data) is sent, which minimizes the load on the network and allows the data repository to enforce and maintain access control [capacity/efficiency advantage]
    - Can be implemented quickly [time to market advantage]
    - Short learning curve to implement; already understood as it is the way the World Wide Web works now [time to market advantage]
    - Intermediaries (e.g. proxy servers, firewalls) can be inserted between clients and resources [capacity advantage]
    - Statelessness simplifies implementation – no need to synchronize state [time to market advantage]
    - Facilitates integration (mashups) of RESTful services [time to market advantage]
    - Can utilize the client to do more work (the client being an untapped resource) [capacity advantage]

2. Some of the disadvantages of REST include:

[3] Internet Link: http://www.ietf.org/rfc/rfc2616.txt
[4] Tim Berners-Lee, Web Design Axiom O.

- Servers and clients implementing/using REST are vulnerable to the same threats as any HTTP/Web application
- If the HTTP commands are used improperly or the problem is not well broken out into a RESTful implementation, things can quickly resort to the use of Remote Procedure Call (RPC) methods and thus have a non-RESTful solution
- There are very few recognized standard libraries which results in additional work on the developer (versus using a standardized protocol with predefined libraries)

C. Assumptions for this Paper

For the purposes of this paper and for the REST implementation guidance, the following assumptions are made:

i. REST implementations rely on the use of the HTTP/HTTPS protocol.
ii. An enterprise service (e.g. PKI, Active Directory) is being utilized for authentication.
iii. An enterprise service (e.g. authorization service, LDAP) is being utilized for authorization. The intent is that the data owner, with input from the necessary authoritative attribute sources, is the final authority for the release of information under their purview. In other words, the authorization service makes the authorization decision (with input from the agency's respective authoritative sources, rules and guidelines) and the application then enforces the access control decision.
iv. Developers are following a 'pure' use of REST (e.g. GET requests are expected to not modify data, POST requests are expected to modify data, PUT requests are expected to create new data, and DELETE requests are expected to delete data) in their implementations. REST solutions do not follow a REST-Remote Procedure Call [RPC] hybrid implementation[5].
v. Other standards are only used when a task cannot be accomplished with a purely RESTful approach. REST implementations should not depart from a 'pure' RESTful implementation (this would create incompatibilities).
vi. Custom solutions are not desired. Preference is for built-in standards-based solutions.

[5] Richardson, Leonard and Sam Ruby, "RESTful Web Services", O'Reilly Media, Sebastopol, CA., ISBN 978-0-596-52926-0, 2007. pp.16-18.

III. Guidance for using GET, PUT, POST, DELETE and HEAD

A few key concepts to understand before implementing HTTP methods include the concepts of safety and idempotence.

A *safe* method is one that is not expected to cause side effects. An example of a side effect would be a user conducting a search and altering the data by the mere fact that they conducted a search (e.g. if a user searches on "blue car" the data does not increment the number of blue cars or update the user's data to indicate his favorite color is blue). The search should have no 'effect' on the underlying data. Side effects are still possible, but they are not done at the request of the client and they should not cause harm. A method that follows these guidelines is considered 'safe.'

Idempotence is a more complicated concept. An operation on a resource is idempotent if making one request is the same as making a series of identical requests. The second and subsequent requests leave the resource state in exactly the same state as the first request did.[6] GET, PUT, DELETE and HEAD are methods that are naturally idempotent (e.g. when you delete a file, if you delete it again it is still deleted).

The importance of safety and idempotence is that it allows users to use HTTP and know that their actions will not change the underlying resource. Of course, this is dependent on uniform usage of the four HTTP verbs. REST relies heavily on this uniform usage so that there are no unintended consequences or unexpected actions. Without a uniform interface, each resource would require a custom interface that understands how that resource expects to send and receive information. This would undermine one of the main goals of REST – simplicity and ease of use.

1. GET: The purpose of the GET command is to retrieve a representation of a resource. GET should never cause any side effects due to its use, in other words it should be used in a safe and idempotent manner. The risk of abusing GET (e.g. not using it in a safe and idempotent manner) is that resources will be changed and / or unintended consequences could alter or affect resources in undeterminable ways. It is poor programming practice to use GET to provide side effects.

2. PUT: The primary purpose of the PUT method is to update, replace or create a new representation of a resource. It is recommended that PUT is used to create new resources only when clients can decide URIs of resources (otherwise use POST).[7] The reason for this is to allow for clients to maintain URI conventions and security or filtering restrictions based on the URI patterns.

3. POST: The POST method is used to make changes, to create and update resources. In a POST request the URI is not necessarily specified and if it is, it represents the resource that will handle the

[6] Allamaraju, Subbu. "RESTful Web Services Cookbook," O'Reilly, 2010, pp. 102-103.
[7] Allamaraju, p. 18.

request vs. the actual entity enclosed with the request (as is done with PUT). The W3C defines POST as a method to cover the following functions[8]:

- Annotation of existing resources
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
- Providing a block of data, such as the result of submitting a form, to a data-handling process
- Extending a database through an append operation

4. DELETE: The DELETE method removes a resource. The server should return a response indicating the success or failure of the operation. Responses to the DELETE method are not cacheable.

5. HEAD: The HEAD command is used to retrieve the same headers as that of a GET response but without any body in the response. The method returns the same response as the GET function except that the server returns an empty body.[9]

The key point to remember is that in order to ensure that an operation / implementation is RESTful, these methods must be used as they were defined in RFC 2616, the HTTP 1.1 specification. If implementations abuse these methods, they not only depart from RESTful behavior, but also jeopardize the application's ability to interoperate with other RESTful capabilities.

IV. Significant Items of Interest

A. Confidentiality, Integrity and Availability

With any development effort, the issues of Confidentiality, Integrity and Availability (CIA) should be addressed. It is advantageous to consider each from the onset of capability development.

With respect to availability, it is assumed that each system will have adequate capability to meet the availability requirements imposed by its respective customers. It is also assumed that corporate services (e.g. authentication and authorization) are robust enough to meet all of the needs of the systems that rely on them. Any inadequacy with corporate services should be documented and sent to the appropriate responsible party.

[8] From Section 9.5 of RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html)
[9] Allamaraju, p. 266.

That leaves confidentiality and integrity as the main focus with implementation of REST. These areas will be addressed throughout the remainder of this paper.

B. Scenarios

Several scenarios will now be discussed to clarify the proper implementation of REST. The first three items will address networking configurations representative of many systems. That will be followed by two use cases demonstrating how information can flow among the various networking configurations.

These scenarios are building blocks. There may be cases where these scenarios are blended (e.g. combinations of scenarios 1 and 2). These scenarios are not an attempt to provide an exhaustive set, but rather a representative set of typical situations.

1. Scenario 1 - Everything exists in the same domain.

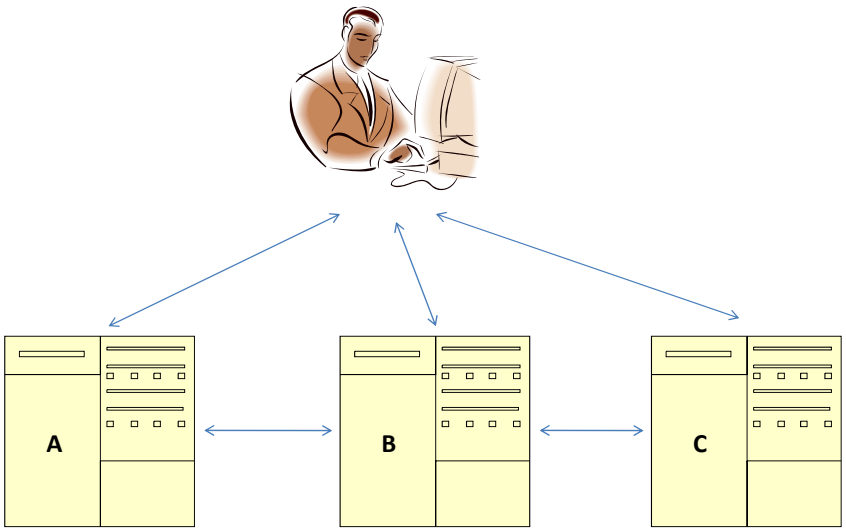


Figure 1. Client has unrestricted access to each server.
Servers can communicate to each other directly.

The first networking configuration is depicted in Figure 1. In this scenario, each server can communicate with each other and the client (user) has access to each server individually. In this case, the client has unrestricted IP access to each server. All systems exist in the same domain. Data aggregation[10] can occur both at the client and at each server; thus user authorization and data protection (for both confidentiality and integrity) is very difficult – it has to be done at both the server and client level.

[10] Data aggregation is the compilation of disparate data sets into a consolidated data set.

2. Scenario 2 - Accessing internal domains.

In this case, the user has a direct connection to each end point. Data aggregation occurs only at the client. This scenario can represent separate servers each with their own mission (e.g. different business units such as Finance and Public Affairs) or data classification (e.g. Unclassified [Server A], For Official Use Only (FOUO) [Server B], and Classified [Server C]). In this situation, authorization and data protection occurs at each server. Data integrity and confidentiality issues are isolated to the client's browser/interface and the transmission path between the client and the servers.

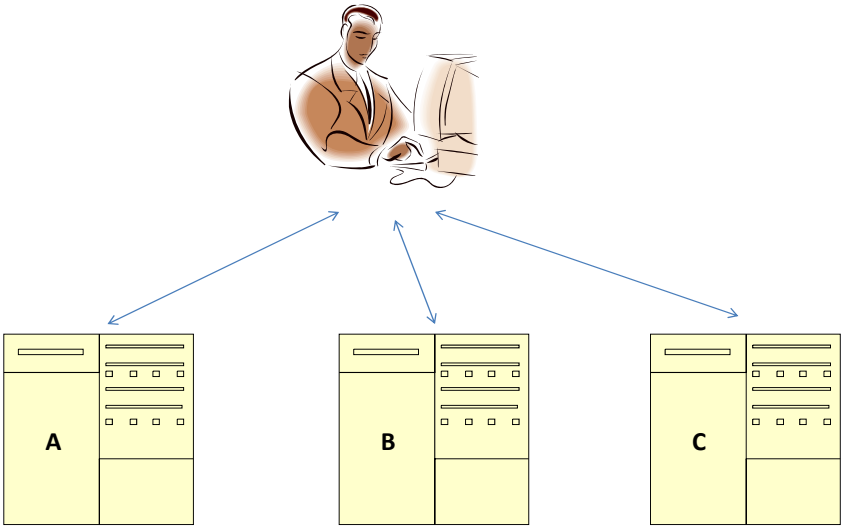


Figure 2. Client has unrestricted access to each server individually. Servers cannot communicate to each other directly.

3. Scenario 3 – Federated search / access on behalf of the user.

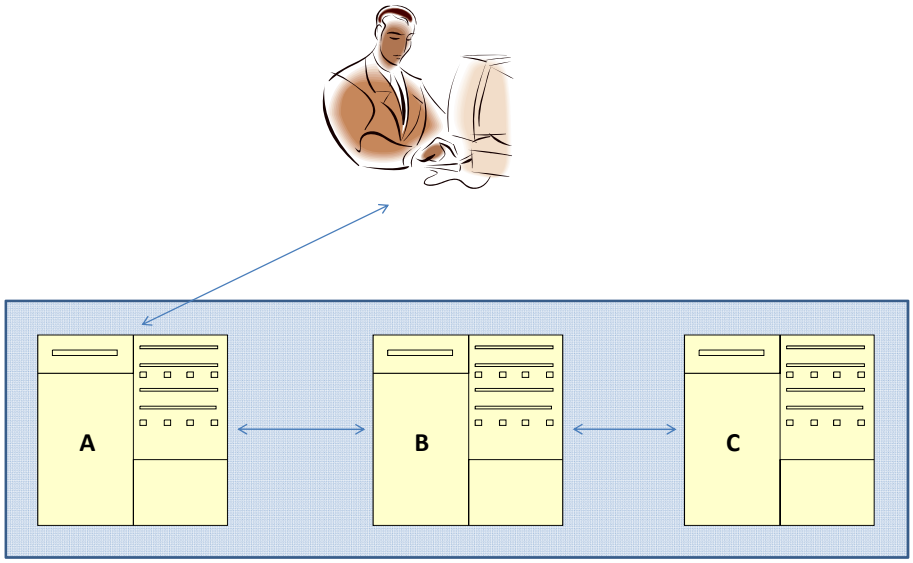


Figure 3. Client accesses one server. Client server acts as a proxy for the user.

In Figure 3, server A, the only server that the client has direct access to, acts as the proxy for the user. Server A executes searches on servers B and C on behalf of the user. The user is authenticated and authorized at server A. Server A can either pass the user's credentials (chained trust) on to servers B and C, where the user's identity is passed on (they are not re-authenticated by server B and C) and servers B and C perform their own authorization check for the passed identity; or Server A can filter the results from servers B and C to what the user is authorized to view. In the Figure 3 situation, the data aggregation occurs at server A.

In situations where server C needs to send sensitive information to the client (via servers A and B), XML encryption[11] should be used to secure the data in transmission, as the client (user) does not have direct access to server C. This scenario is not in line with the traditional REST model, as the user does not have direct access to the data source. SOAP may be more appropriate to use in this type of situation.

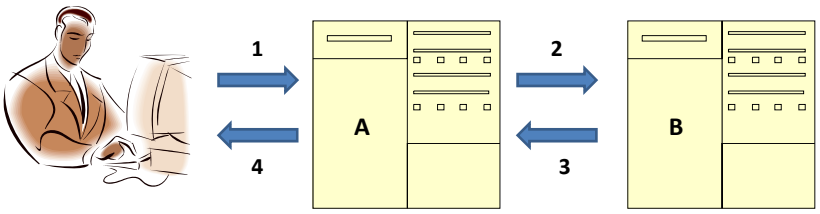
Data integrity and confidentiality issues exist between the servers and the client as well as at the aggregation point (server A).

[11] See http://www.w3.org/TR/xmlenc-core/ on the internet. Last accessed on 10/5/10.

C. Use Cases

Two use cases will now be presented to further delineate the security implications of using REST.

1. Use Case 1:



Key:
1 – "Give me information about X."
2 – "B give A information about X for client (user)."
3 – "A, Here is information about X"
4 – "User, Here is information about X."

Figure 4. Use Case 1: Federated Search

Figure 4 presents Use Case 1 where all client requests go through server A and the client does not have direct access to server B. The advantages and disadvantages of this scenario are outlined in the table below:

Advantages	Disadvantages
1. Simpler client interface	1. Chained trust (AuthZ)
2. Data aggregation guard at Server A	2. Intermediaries
3. Less client work to conduct a search	3. Server A has knowledge of the search parameters

Use Case 1 depicts a typical scenario where server A acts as an intermediary. Often, in large networks, it is difficult to have direct point to point connections. The challenge, in this situation, is how to protect sensitive information in transit such that it is not available to those who are not authorized to access it (e.g. the situation if the client and server B are authorized a higher level of access than server A). Intermediaries, server A in this case, can impact the confidentiality and integrity of the data being sent from server B to the client. This situation requires the use of

encryption to maintain the confidentiality and integrity of the data as it moves from server B to the client.

With REST, the preferred method to prevent unauthorized access in this scenario would be to utilize encryption, such as XML encryption, to protect the information sent from server B to the client.

2. Use Case 2:

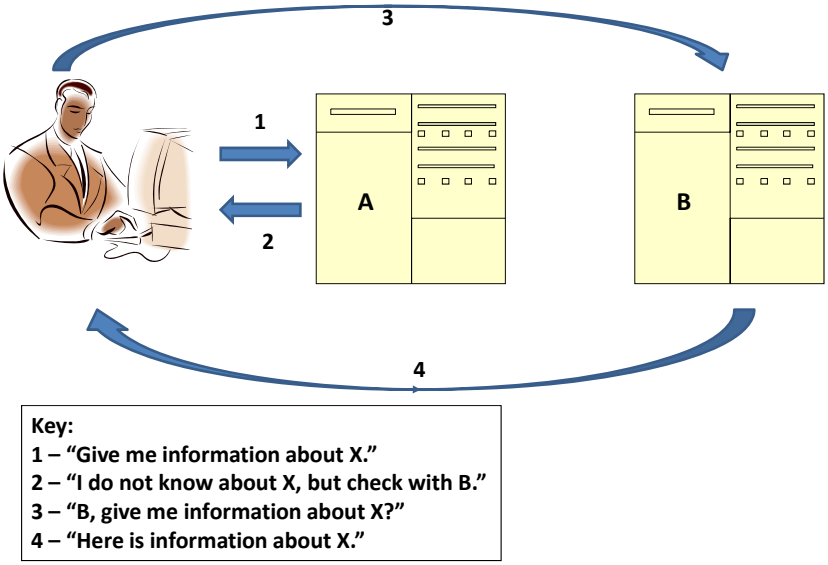


Figure 5. Use Case 2: Client re-direction.

In the scenario in Figure 5, server A could either not have knowledge of the requested action or be untrusted. Either way, server B (trusted) is invoked to return the requested information to the client. In this case, the client ends with a direct connection to the resource, thus facilitating proper authorization at server B.

Advantages	Disadvantages
1. Direct connection to client	1. Server A has knowledge of requested search
2. Client could have better data access	2. No aggregation guardian
3. Data source – Authorization directly to client	3. Client may have to perform "potentially sensitive computation"
4. Server A can be untrusted	4. Multiple client requests (heavy processing)

Use Case 2 is a valid situation where REST can be used. The client has direct access to the data source. A reference to the actual data can be sent from server B to the client and then the client can go directly to server B to retrieve the data. Confidentiality and integrity are easily maintained in this situation.

D. Service/Security Chaining

Chained trust is challenging for web service implementations and the situation is no different with REST. The chained trust situation is exemplified in Use Case 1, Figure 4. In this situation a user's credentials are passed through the main computer (server A) to the remote computer (server B) so that the remote computer can execute the requested operation. The chain of trust in this example is from the user to server A to server B. In this case, since server A cannot execute the requested operation, server A does not need to know the user's credentials. The risk in this chain of trust is that server A could alter the request, either by changing the client's information or altering the information being returned to the client.

There are a few ways to counter the threat with chained trust. The steps are as follows:

1. Use PKI (both user and server).
2. Enforce mutual authentication between servers.
3. Pass the user's identity from server to server.
4. Perform an authorization validation at the data source.

With REST, server B could return a URI to the requested information. This URI would be passed to server A which would then pass it on to the requesting user. For scenarios 1 and 2, the user could then access the information directly with the supplied URI.

For scenario 3, the user does not have direct access to the information resources (direct access via a URI is not possible in this scenario). In this situation, XML encryption should be used to encrypt the information being sent from server B to the user.

E. Potential Vulnerabilities

As REST is a web-based technology, it is susceptible to the 'typical' system and web (HTTP) vulnerabilities. Some of these vulnerabilities include:

1. System Vulnerabilities:
    a. Poor coding practices
    b. Configuration issues (e.g. weak encryption)
    c. Lack of proper test and evaluation

2. Web Application Vulnerabilities:
    a. Session hijacking

b. Cross-Site Scripting (XSS)
c. SQL Injection (see SNAC Tech Note on SQL Injection)
d. Format String Vulnerabilities
e. Inadequate authentication/authorization methods
f. Cross Site Request Forgery (CSRF)
g. Access control policies

The Systems and Network Analysis Center (SNAC) has already published guidance on how to address some of the typical web service vulnerabilities. A summary of web application security vulnerabilities is available on the SNAC web page. Additional guidance for system and application configurations is available on the NSA web site at http://www.nsa.gov/ia/guidance/index.shtml or http://www.nsa.gov/ia/guidance/security_configuration_guides/index.shtml. It is strongly recommended that all application development follow the available guidance in order to protect against these vulnerabilities, and that those responsible for software testing[12] ensure that tested applications are adequately protected from these known vulnerabilities.

Data aggregation in the browser is an issue for any web service that gathers data from more than one source. The browser has no means to address data aggregation from an authorization perspective. In this situation, authorization decisions must be performed by each data source and the resulting data returned to the user's browser. As this issue is not specific to REST, we will not address it further in this paper.

The CSRF vulnerability needs some additional explanation as it is important to REST implementations involving the interaction of a web-browser and a web-server.

F. Cross Site Request Forgery:

Cross site request forgery (CSRF) attacks attempt to force an authenticated user to execute functionality without their knowledge.

> "The attack works by including a link or script in a page that accesses a site to which the user is known (or is supposed) to have been authenticated.[13] For example, one user, Bob, might be browsing a chat forum where another user, Mallory, has posted a message. Suppose that Mallory has crafted an HTML image element that references a script on Bob's bank's website (rather than an image file), e.g.,

[12] Suggested testing guide (Internet link) – http://www.owasp.org/index.php/Category:OWASP_Testing_Project, accessed on 10/4/10.
[13] Shiflett, Chris (December 13, 2004). "Security Corner: Cross-Site Request Forgeries". Php|Architect (via shiflett.org). Retrieved 2008-07-03.

> <img src="http://bank.example/withdraw?account=bob&amount=1000000&for=mallory">
>
> If Bob's bank keeps his authentication information in a cookie, and if the cookie hasn't expired, then the attempt by Bob's browser to load the image will submit the withdrawal form with his cookie, thus authorizing a transaction without Bob's approval."[14]

It is important to note that CSRF attacks execute functionality, but are unable to interact with the response from the targeted server. In the above example, if the bank URI responded with HTML that required the user to click on a randomly generated link prior to withdrawing money, then the attack would fail as the CSRF attack cannot interact with the exploited site's response.

While most examples of CSRF include attacks targeting GET based actions and some defense advice involves moving functionality to POST actions, this is not sufficient as CSRF attacks targeting POST actions are as trivial as those targeting GET actions.

REST is stateless at its core and thus inherently vulnerable to CSRF attacks. In order to prevent CSRF attacks targeting a REST based API, two potential approaches are recommended.

The first method involves setting custom headers for each REST request such as X-XSRF-Header. The value of this header does not matter; simply the presence should prevent CSRF attacks. If a request comes into a REST endpoint without the custom header then the request should be dropped.

HTTP requests from a web browser performed via form, image, iframe, etc are unable to set custom HTTP headers. The only way to create a HTTP request from a browser with a custom HTTP header is to use a technology such as Javascript XMLHttpRequest or Flash. These technologies can set custom HTTP headers, but have security policies built in to prevent web sites from sending requests to each other unless specifically allowed by policy. This means that a website www.bad.com cannot send a request to http://bank.example.com with the custom header X-XSRF-Header unless they use a technology such as a XMLHttpRequest. That technology would prevent such a request from being made unless the bank.example.com domain specifically allowed it. This then results in a REST endpoint that can only be called via XMLHttpRequest (or similar technology).

It is important to note that this method also prevents any direct access from a web browser to that REST endpoint. Web applications using this approach will need to interface with their REST endpoints via XMLHttpRequest or similar technology.

---

[14] Cross-site request forgery, from Wikipedia, http://en.wikipedia.org/wiki/CSRF#cite_note-Shiflett-0. Retrieved 2010-9-22.

This method attempts to limit the way in which a REST endpoint can be called by using security mechanisms built into other web technologies to prevent CSRF attacks on a REST endpoint without forcing the REST system to establish session state.

The second approach involves protecting REST endpoints against CSRF attacks by establishing session state. Without establishing session state, the API endpoints cannot determine if the request came legitimately from a user or from a user being forced to submit the request. In order to resolve this issue, it is recommended that some amount of state be maintained between the web-server and the web-browser, even though it violates the initial principles of REST. With this state maintained it becomes possible to implement a CSRF token endpoint in the API such as http://site.com/get_csrf_token. Once the endpoint is called, the web-server can issue a randomly generated CSRF token that the client must submit with every API request.

Traditional defenses include the use of a CSRF token that is generated for each action, then marked in the user session and submitted with each important website action. This essentially forces a sequential ordering of actions on a website. For example, Bob would not have his CSRF token until he visited the page http://bank.example.com/setup_withdraw which then identifies his CSRF token to the browser and the web-server. In the above example, if we imagine that the CSRF token was 23, then the attack would not work unless the value 23 was submitted with the link (e.g. http://bank.example/withdraw?account=bob&amount=1000000&for=mallory&csrf_token=23).

It is important to note that because CSRF is a blind attack and cannot read content from an attack, CSRF protections need only be applied to endpoints that will modify information in some way. This means that if a true RESTful implementation is used, the CSRF protections described above only need to be applied to requests using the POST, PUT or DELETE verbs. This assumes that GET is used as intended by RFC 2616. If a non-standard RESTful implementation is used, the need for CSRF protection on an endpoint will vary by application.

G. Encryption

The need for encryption has already been seen in the use cases previously presented. Encryption should always be used to transfer data or sensitive information. The two main types of encryption that can be used with REST are transport (e.g. HTTPS) and data (e.g. XML) encryption.

1. Transport (HTTPS) Encryption

HTTPS is network layer encryption that utilizes HTTP over Secure Socket Layer (SSL). HTTPS can be used for encrypting point to point (single network hop)

communications. If there is an intermediary, then XML encryption should be used. Transport Layer Security (TLS)[15] is the upgrade to SSL version 3. It is recommended that TLS version 1.2[16] or newer be utilized (vs. SSL v1.2), secure ciphers are utilized and prior versions of TLS and SSL are not allowed.

Case 2, Figure 5 is a good example of where HTTPS can be successfully used to secure the client and server communications. In this case, the client and server are directly connected.

2. Data (XML) Encryption

In Case 1, Figure 4, the client has to go through server A to get data from server B. In this case, server A is an intermediary. Server A needs to know which client is communicating with which server and thus must understand part of the communications. If HTTPS was used to communicate between server B and the client, server A would need to decrypt the message to determine the intended recipient. Data encryption solves the intermediary problem and provides a means for encryption across multiple network hops.

V. Recommendations

A. General Recommendations:

Since REST relies heavily on the HTTP protocol, many of the tools used to secure HTTP can also be used to secure REST. Some examples[17] of tools and recommended best practices include:

1. Use of HTTPS. All data must be sent over HTTPS.
2. Use of PKI for authentication (if PKI cannot be used, HTTP Digest Authentication can be used with HTTPS/TLS).
3. Use of an authorization service (e.g. LDAP).
4. EVERYTHING needs to be identified as a unique resource (and has a URI) so that it is directly addressable.
5. Resources expose a standard interface using only GET, PUT, POST, DELETE.
6. Link resources together using URIs.
7. Allow representations in multiple formats (e.g. HTML, XHMTL, XML, JSON) to ensure interoperability with other systems.

[15] See http://en.wikipedia.org/wiki/Transport_Layer_Security on the Internet. Last accessed on 10/5/10.
[16] See http://netinfo.proj.nsa.ic.gov/www.faqs.org/rfcs/rfc5246.html on NSANet. Last accessed on 10/5/10.
[17] Ozone Project. "Designing REST Services.", https://wiki.itd.nsa/img_auth.php/b/b2/OZONE_DesigningRestServices_1.8.pdf, p. 23.

8. Utilize error codes. It is highly recommended that error codes are returned whenever an error is encountered. This will help the user determine if the error is due to user action, web service or a server issue. A cautionary note here is to not provide too much information (such that it would provide an adversary an advantage). Successful error codes/messages are a balance between enough information and security.
    a. For errors due to client inputs, return an error representation with a 4xx code.
    b. For errors due to server issues, return a 5xx code.
9. Design URIs to be persistent. If a URI needs to change, honor the old URI and issue a redirect to the client.
10. Caching should generally be avoided where possible and sensitive data should never be cached.
11. Sensitivity of the URI: When developing REST solutions, care needs to be taken not to create URIs that contain sensitive information.

B. Additional best practices:

1. The data owners (and their associated repositories) should remain the policy decision point (PDP) for allowing access to data. The requester should be authenticated and authorized prior to completing an access control decision. All access control decisions shall be logged.
2. Code as if protecting the application.
3. There are several additional resources that exist. They include:
    a. "RESTful Web Services Cookbook" by Subbu Allamaraju (ISBN 978-0-596-80168-7).
    b. "RESTful Web Services" by Leonard Richardson and Sam Ruby (ISBN 978-0-596-52926-0).

C. Consideration for use of SOAP:

1. SOAP should be used for situations where REST will not provide adequate security, audit traceability, etc. REST may not always be the most efficient approach to use. For these situations, SOAP should be considered.
2. For legacy applications already utilizing SOAP, do not change to REST just for the sake of using REST. If a new interface is required, then the use of a RESTful interface could be considered. It is not difficult to put a RESTful façade over a SOAP interface.
3. REST and SOAP can peacefully co-exist. For example, REST could be used for the user interface and SOAP could be used for the server to server communication.
4. Encrypt to the endpoints. When encryption through intermediaries is required, XML encryption shall be used. Due to the confidentiality and

integrity issues presented in Case 2 by the use of server A as an intermediary, it may be more advantageous to utilize SOAP for the transactions, as SOAP provides a standard for secure transmissions (e.g. XML encryption with WS-Security). Where SOAP provides standard means to utilize XML encryption, REST would require a custom solution. In this case, SOAP should be used.

VI. Summary

There has been a lot of concern related to REST security and the perception that REST lacks strong security tools/controls. The reality is that, if implemented according to the HTTP RFC, REST is no more vulnerable or secure than any other web application/standard. As mentioned in section IV-e, all web services are susceptible to a number of vulnerabilities – whether they use REST or SOAP. The key in countering these vulnerabilities is in the implementation of the service. While SOAP provides a number of standards to assist with security, if they are not properly implemented they will not work. Likewise, if REST is not implemented in accordance with the HTTP RFC, it will not perform properly, may not interoperate, and will be more difficult to secure. Security will be a factor in development/deployment/release decisions. The risks need to be identified, understood and evaluated. This paper tries to help identify and explain the security risks (positive and negative) with REST, to facilitate development of more robust REST solutions.

The advantage of REST is identical to that of the Web – any HTTP client can talk to any HTTP server. This ease of connection is what has allowed the web to grow quickly and support many different hardware and software platforms. The desire to utilize this philosophy in the interaction of software services is obvious. The challenge for agencies is to find a way to do that without losing their capability of ensuring the proper access to and release of data and information.