Communications of the IIMA

Volume 13 | Issue 1 Article 5

2013

Defense Against REST-based Web Service Attacks for Enterprise Systems

Hsun-Ming Lee
Texas State University

Mayur R. Mehta
Texas State University

Defense Against REST-based Web Service Attacks for Enterprise Systems

Hsun-Ming Lee
Texas State University, San Marcos, USA
sl20@txstate.edu

Mayur R. Mehta
Texas State University, San Marcos, USA
mm07@txstate.edu

ABSTRACT

In recent years, Representational State Transfer or REST-based Web Services have become popular for building Web systems. They have become an integral and critical part of information systems to facilitate and integrate the business processes across the enterprise. However, the simplicity of a REST-based implementation has caused the neglect of its systematic security threat analysis and design. One of the issues of systems built with REST services integration is their susceptibility to JSON input attacks. Such attacks could compromise the integrity of critical data in enterprise business processes. We analyze such a security issue in this paper. Some mechanisms used to secure Web sites and servers, such as encryption via HTTPS, static source code analysis, and input validation, can be integrated to defend against the attack.

Keywords: Information security, REST, web services, input injection, enterprise information systems, JSON

INTRODUCTION

In the past, most enterprises adequately assured information security through operation systems dealing with access controls on resources such as files and network connections. However, according to Pistoia and Erlingsson (2008), attacks may happen at higher levels of abstractions and may target the internal behavior of applications. Recently, enterprises have quickly been moving toward the development of decentralized, flexible, and layered application architecture to reach their clients on the Web or mobile environments. In order to successfully make this transition, they must treat application level security as a major threat (Jain & Shanbhag, 2012; Kundu, Sural, & Majumdar, 2010; Shar &Tan, 2012).

The application level security has become even more critical since the emergence of Web 2.0. Due to its openness and ease of adoption, there has been an explosion of public API’s (Application Programming Interface) allowing developers to call functions and data from multiple diverse services to create new applications (Werts, Mikhailova, Post, & Sharp, 2012). Representational State Transfer or REST-based Web Services, such as Google Geocoding API and Facebook Graph API, are among the most popular today. It is not surprising that application developers have started to take advantage of this technology to facilitate business processes in enterprise (Su & Chiang, 2012). In today's business environment, business processes are more likely to be dynamic, distributed, and collaborative, making it necessary to adapt business processes more often, and integrate processes across organizational boundaries with business partners. Further, organizations also need to expose business-critical functionalities, embedded within millions of lines of mainframe code developed over the past four decades (Mitchell, 2006), and integrate these functionalities with new Web-based or mobile user interfaces. The REST technology provides a cost-effective and an efficient alternative to support tight business process integration.

The simplicity and adoptability of the REST-based web services, however, come with a price. A REST implementation provides no pre-defined security protection methods. Application developers utilizing such services as system components must diligently ensure that information integrity in all of maintained business records is not compromised. A significant monetary loss is very likely should such integrity is compromised. This paper is aimed at presenting vulnerabilities of REST-based Web services built for distributed applications in enterprise information systems. In particular, we focus on the integrity of application output of such business systems. If a security flaw exists, hackers can register as legitimate users and carry out attacks through data input or "input injection attacks." This paper discusses a possible defense against these attacks.

To provide a necessary foundation for understanding security vulnerabilities in distributed applications, the paper first provides a brief overview of the technology, including Web Service security, in the next section.

TECHNOLOGY BACKGROUND

<u>Web Services and the MVC Framework</u>

Application architectures such as Model-View-Controller (MVC) framework and its variations that decouple presentation and business logic has become a de-facto blueprint that application developers use to design and implement flexible, scalable, and maintainable Web solutions. Thus, modern enterprise information systems that are implemented on the Internet or Intranet frequently use the MVC framework. In an MVC framework, Models (M) are meant to serve as a computational abstraction of real entities. For example, a model of a product in an e-commerce application contains the identity and quantity of the inventories that are available for the product. Views (V) present or render the information that is contained in a model. A controller (C) component controls the application flow. It accepts input from the user, interprets the input, and invokes the appropriate service in response to the input. For example, a Web service activated by a controller provides the functions for finding all products or a product by a product identity number.

The traditional or SOAP-Based Web services implementation at the minimum involves a collection of three Extensible Markup Language (XML) standards to support communication between interacting services. These include standards to describe services, standards to publish services, and standards to discover services. The three main standards that enable implementation of traditional Web services are the Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). The implementation of Web service based systems may require a significant investment by organizations, as indicated in Wachovia Bank's example (Margulius, 2006), and will most likely also necessitate a change in the organizational IT culture and practices (Brown, Delbaere, Eeles, Johnston, & Weaver, 2005).

Web Service Security

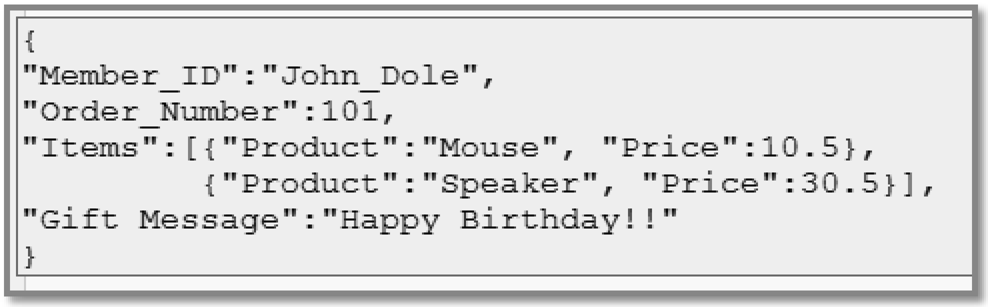
SOAP-based Web services are prone to all of the same attacks that can be launched on a standard Web application, such as SQL injections, capture and replay attacks, buffer overflows, denial-of-service attacks, and improper error handling (Jaamour, 2005). To address the specific concerns of privacy, authentication, and integrity for Web services, Jaamour points out that WS-Security is a standard way of securing a single message by applying Username Tokens, XML Signatures, and XML Encryption. Several WS-Security implementations have been suggested. An IBM team developed a security service bundled within WebSphere Application Server (Makino, Tamura, Imamura, & Nakamura, 2004). XML transcripts transmitted between two colleges were secured by implementing WS-Security using Bea WebLogic Workshop (Lim, Sun, & Vila, 2004).

The WS-Security implementation on Web servers provides basic Web service security with performance costs. There are potential bottlenecks in the XML parsing and the public key operations for encryption. Additionally, a straight implementation requires performance improvement (Makino et. al, 2004). Even protected by WS-Security, a SOAP-based Web service was still vulnerable to various types of newly found attacks including XML external entity attacks, XML bombs, malicious SOAP attachment, and XPath injections (Jaamour, 2005). Thus, to ensure security and integrity of any web services, it is essential to keep updating their security measures and safeguards.

REST-based Web Service and JSON

Compared to SOAP-Based Web services, which face many limitations discussed above and are somewhat difficult to implement, a REST-based approach to Web services is much easier to implement since its design simply relies on the HTTP protocol. It uses (a) URIs (Uniform Resource Identifiers) to identify resources; and (b) the GET, PUT, POST and DELETE actions to retrieve, update, create, and delete the resources remotely through Web servers. In addition, JavaScript Object Notation (JSON), a text-based data interchange format that is completely language independent and provides significant performance gains over XML due to its light weight nature and native support for JavaScript (Ying & Miller, 2013), is an excellent way to transport/exchange messages between services as well as between client and servers.
According to the JSON website, JSON is simply built on two data structures:

- Data to be exchanged are enveloped as an unordered set of name/value pairs. Each data item is represented by a name followed by a colon and then its value.
- Multiple data items may be packaged in a single object as an array of ordered name/value pairs, with each pair separated by a comma. The array is enclosed between a left opening bracket and ends with right closing bracket. An example is provided below:



```
{
"Member_ID":"John_Dole",
"Order_Number":101,
"Items":[{"Product":"Mouse", "Price":10.5},
         {"Product":"Speaker", "Price":30.5}],
"Gift Message":"Happy Birthday!!"
}
```

Because of its simplicity of implementation and lightweight nature resulting in improved performance, REST-based Web services that use JSON objects as data transports have gained popularity amongst developers. JSON is beginning to become a clear choice among developers and application architectures for mainstream data applications (Severance, 2012). Severance also points out that programmers use it extensively to connect two servers communicating via Web services. In this paper, we focus on the security issues where JSON strings are used to represent the request and response messages in a REST-based Web services.

Data transfer security in a single REST-based Web service can be easily achieved by encryption using HTTPS (Kennedy, Stewart, Jacob, & Molloy, 2011). In a server providing sophisticated REST-based Web services, such as Facebook, Google, and Twitter, OAuth is a popular solution for authorizing third-party applications (Shehab & Marouf, 2012), and so the application's user session can obtain a credential once and use it whenever the access to server resources is necessary. It is noteworthy that security measures are optional for implementation and likely overlooked when REST-based Web services are built upon ad hoc additions to legacy applications. For an enterprise system to succeed, it must additionally require appropriate protection that (a) is applied to information from the moment information is created to the moment it undergoes final disposition, and (b) protect against unauthorized alteration, destruction, loss, or disclosure of information (Manago, 2011). It can be argued that the simplicity of the REST-based approach, causing its neglect in systematic security analysis and design (Comerford & Soderling, 2010), makes it more vulnerable to security risks than the traditional Web services. To meet the system requirements, it is essential to investigate thoroughly potential security issues of REST-based Web services built for the enterprise.

THREAT AND COUNTERMEASURE

A simple scenario is presented here to exemplify security threats encountered in an enterprise system that is built upon REST-based Web services. Consider a customer-facing system that allows users to register as customers, browse a catalog, place an order, enter shipment instructions, and pay for the order on the Web. To utilize the legacy systems in an enterprise, this customer-facing web site is connected to many REST-based Web services, including those for managing membership, order and fulfillment processes, and payment (see Figure 1). The system uses firewalls to block illegal connections and user authentication/authorization processes to ensure only registered and authorized users are allowed access to the e-Business site to create and pay for their orders. Even with this strong network-level protection, the system is potentially vulnerable to security flaws of REST-based Web services.

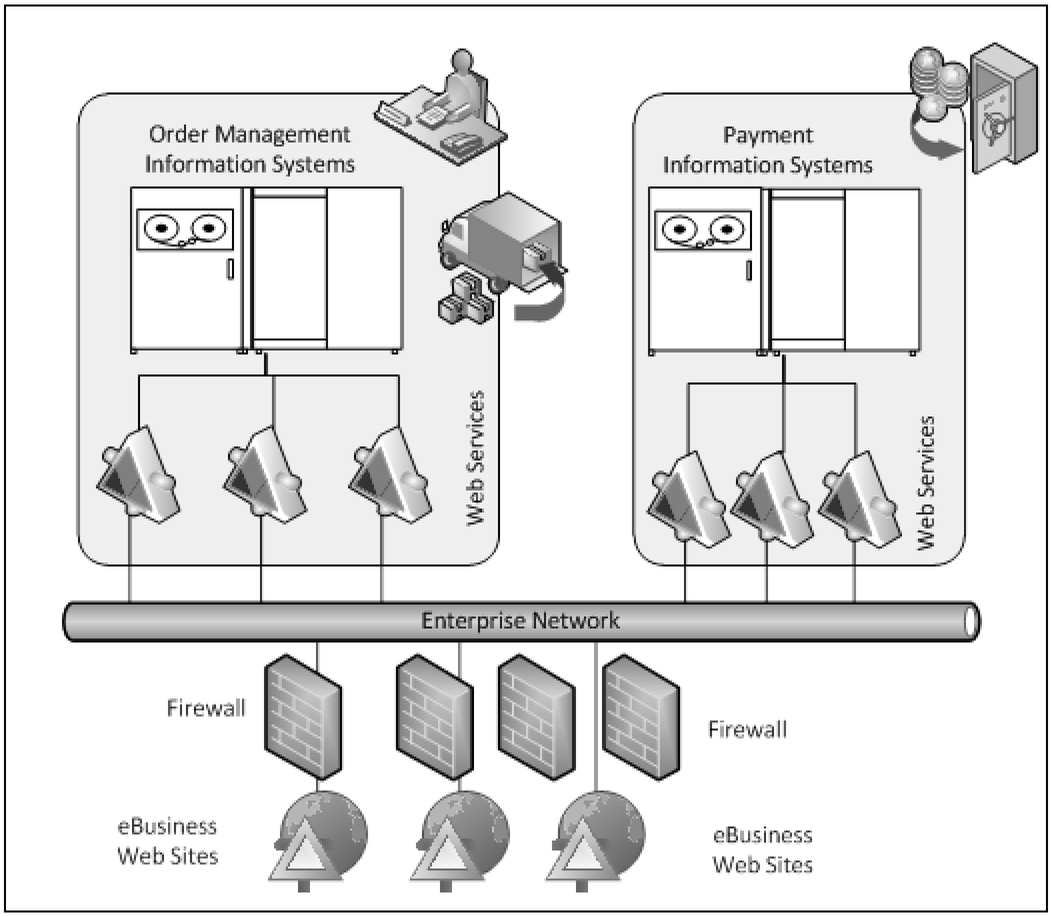
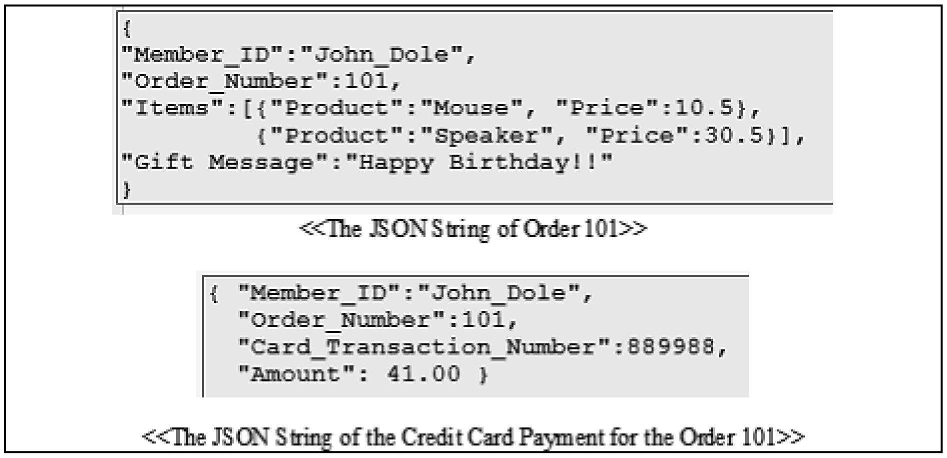


Figure 1: An Enterprise Information System Built by REST-based Web Services.

Threat

We pay special attention to some critical data from the moment it is created to the moment it undergoes final disposition. In this case, order and payment data are entered by users to create an order on a Web site. Then, the data are sent to legacy systems for fulfillment and billing processes. Once, the user has paid for the order and the credit card is charged, the data can be archived and may be eventually be deleted. Thus, the data integrity of an order and its payment should at least be examined carefully when (a) the data are received on the web site; and (b) they are stored on the legacy systems.

The threat comes from an attacker that registers as a customer and places an order through the site. Typically, such sites allow customers to add products to a shopping cart, select a shipping method, enter a gift message for the shipping, and pay the order by a credit card. Assuming that the customer-facing website interacts with legacy enterprise systems via REST-based Web services, this interaction will result in two separate JSON messages being generated, order and payment. These are sent to two separate legacy information systems via REST-based Web services after checkout. Typical JSON messages, created by and sent from a business web site, should look like those shown in Figure 2. It shows the JSON strings of an order and its payment, respectively. The corresponding REST Web services will receive and then convert or parse these strings into appropriate software objects representing an order and a payment for further processing in information systems (see Figure 3 for the results). For the sake of demonstration, the JSON strings shown in Figure 2 are parsed by an Online JSON Parser (http://jsonparser.com/) and shown in Figure 3.



```
{
"Member_ID":"John_Dole",
"Order_Number":101,
"Items":[{"Product":"Mouse", "Price":10.5},
        {"Product":"Speaker", "Price":30.5}],
"Gift Message":"Happy Birthday!!"
}
```

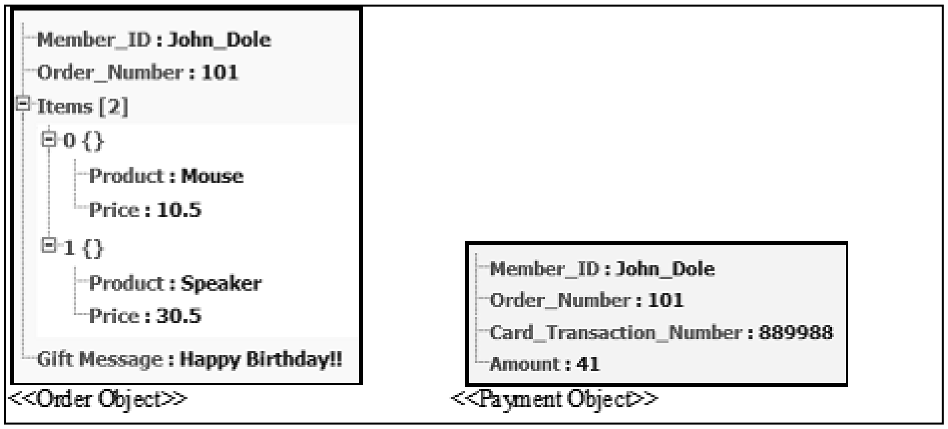
<<The JSON String of Order 101>>

```
{ "Member_ID":"John_Dole",
  "Order_Number":101,
  "Card_Transaction_Number":889988,
  "Amount": 41.00 }
```

<<The JSON String of the Credit Card Payment for the Order 101>>

Figure 2: Sample JSON Strings of an Order and its Payment.

In this example, John Dole could be a possible attacker trying to manipulate the gift message in the order to receive an expensive product and pay much less for it. The JSON string for a manipulated order is illustrated in Figure 4. As you can see, John enters a much more complicated message in the Gift message field to include a string that would mimic an ordered product's name/value pair. Figure 5 shows the result of this JSON injection message when parsed by the receiving Web service. Note that John ordered two products worth $41 through the Web site. His credit card is charged $41. Because of the injected gift field message, the parsed JSON message to be processed by the order fulfillment process shows that John ordered a $1000 computer. Thus, the company is fooled into shipping out a much more expensive product. This illustrates a potential application-level security flaw when using REST-based Web services.



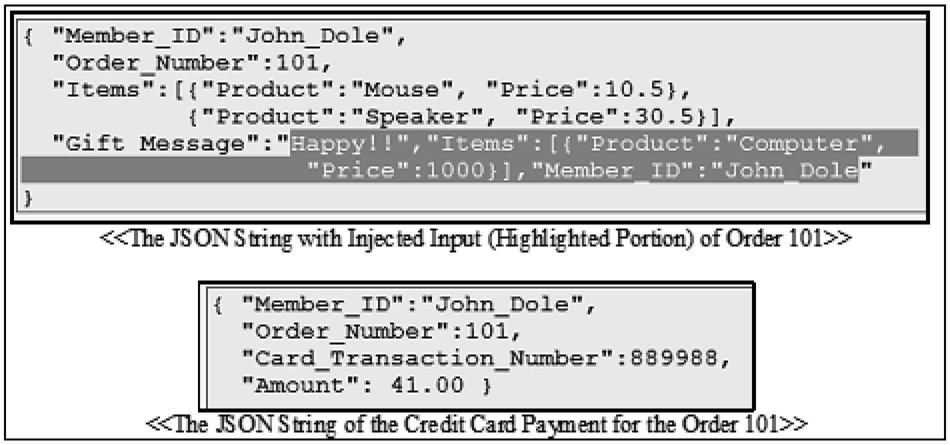
```
Member_ID : John_Dole
Order_Number : 101
⊟ Items [2]
  ⊟ 0 {}
      Product : Mouse
      Price : 10.5
  ⊟ 1 {}
      Product : Speaker
      Price : 30.5
Gift Message : Happy Birthday!!
```

<<Order Object>>

```
Member_ID : John_Dole
Order_Number : 101
Card_Transaction_Number : 889988
Amount : 41
```

<<Payment Object>>

Figure 3: The Order and Payment Objects Generated (from the JSON Strings in Figure 2) by Rest-based Web Services Using a JSON Parser.



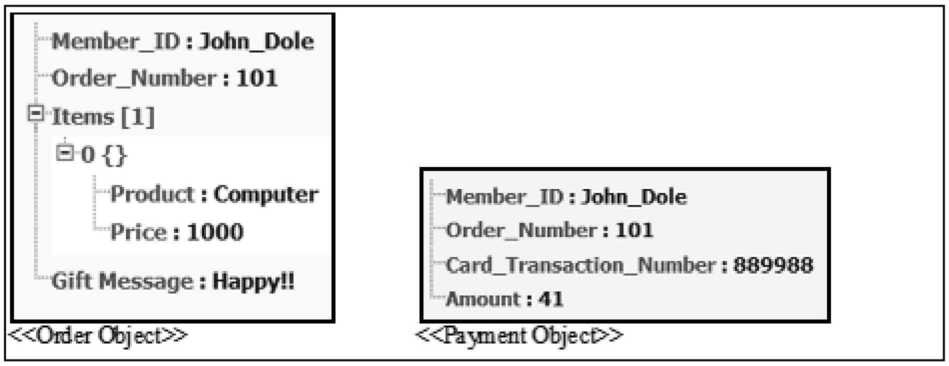
```
{ "Member_ID":"John_Dole",
  "Order_Number":101,
  "Items":[{"Product":"Mouse", "Price":10.5},
           {"Product":"Speaker", "Price":30.5}],
  "Gift Message":"Happy!!","Items":[{"Product":"Computer",
                   "Price":1000}],"Member_ID":"John_Dole"
}
```

<<The JSON String with Injected Input (Highlighted Portion) of Order 101>>

```
{ "Member_ID":"John_Dole",
  "Order_Number":101,
  "Card_Transaction_Number":889988,
  "Amount": 41.00 }
```

<<The JSON String of the Credit Card Payment for the Order 101>>

Figure 4. Sample JSON Injection of an Input Order and its Payment.



```
Member_ID : John_Dole
Order_Number : 101
⊟ Items [1]
  ⊟ 0 {}
      Product : Computer
      Price : 1000
Gift Message : Happy!!
```

<<Order Object>>

```
Member_ID : John_Dole
Order_Number : 101
Card_Transaction_Number : 889988
Amount : 41
```

<<Payment Object>>

Figure 5: The Order and Payment Objects Generated by Rest-based Web Services after a JSON Injection.

<u>Counter Measures</u>

Insiders, including former employee and contractors are often the most motivated and resourceful at breaching the network security of an organization (Ortega, 2007). The vulnerability of the REST-based Web services provides easy targets for internal attackers. By sniffing a JSON string, they can easily figure out its object properties and carry out attacks through the open eBusiness Web sites. What counter measures are available to mitigate such application-level threats?

Two simple counter measures may be put in place to mitigate such security threats. First, the basic input validation can be programmed in the application to prevent a JSON injection even after the attackers have learned the JSON message format. The Web services could validate received the JSON message for unwanted characters such as double quotes, colons, and brackets, before parsing the messages into software objects, However, this technique is somewhat problematic since certain special characters such as colons in time notations need to be allowed in some applications (Valli, 2006). Secondly, to prevent the internal attackers from stealing or modifying the JSON message once this message has entered the enterprise system, the encryption of the JSON string in the internal network should be a high priority, especially if there is a high cost resulted from compromised system data integrity. In a typical system, only a very few internally transferred messages are encrypted due to the high implementation cost and perceived low risks.

Static analysis, examining the text of a program statically without attempting to execute it (Chess & McGraw, 2004), may be necessary to ensure that programmers implement the counter measures appropriately. For a REST web service installed with the HTTPS capability (Wasson, 2012), programmers must code the word “HTTPS” literally in a statement to test and activate the secure message transfer for the clients. Static analysis is crucial since most security attacks exploit either human weaknesses - such as poorly chosen passwords and careless configuration - or software implementation flaws (Evans & Larochelle, 2002). A static code analysis tool could be created to detect and warn overlooked HTTPS activations.

A key to design the tool is the specification of semantic characterization for the known security vulnerability (Bishop & Dilger, 1996). To generate the specification for REST-based Web services, the topology of links between pages (navigation model) is first needed and could be created using the Web Modeling Language (Ceri, Fraternali, & Bongio, 2000). We give an example of WebML composition and navigation specification (see Figure 6) based on the eBusiness website of the Enterprise Information System illustrated in Figure 1.

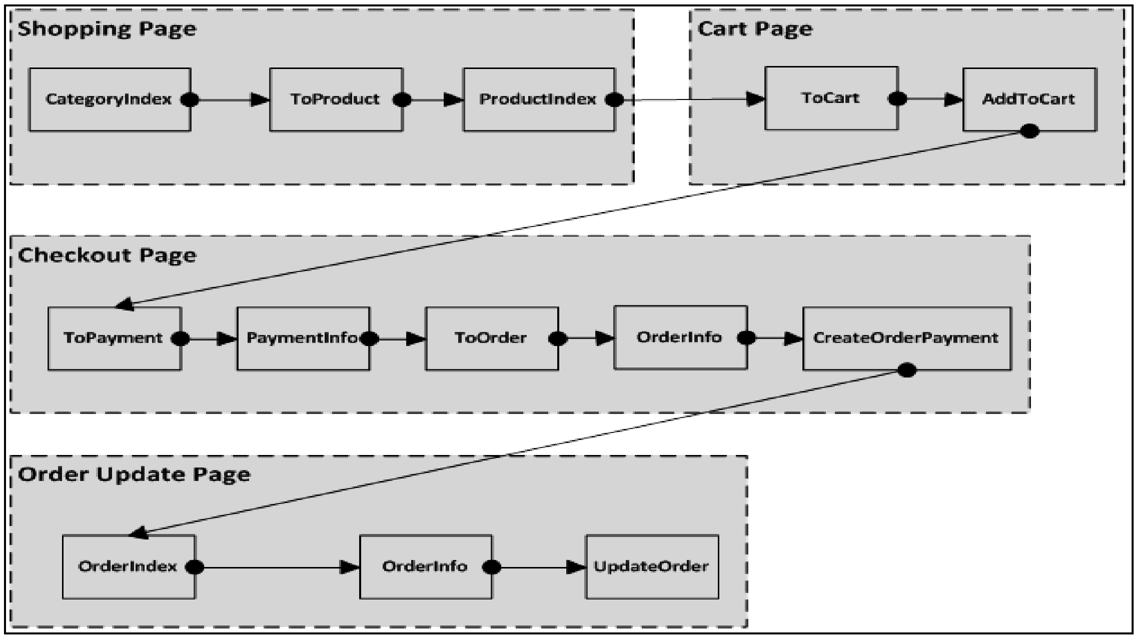


Figure 6: An Example of Navigation Model for an eBusiness Website in the Enterprise Information System.

Next, the page links are used to define a vulnerable pattern of Web service execution (see Figure 7). We use the popular Curl command (http://curl.haxx.se/) to represent the programming code for transferring data with URL syntax in a Web page script. To activate a REST-based Web service, the command must include an action (GET, POST, PUT, or Delete) and an URL formed by a protocol (HTTP or HTTPS), a hostname, and an object name. We describe the vulnerable pattern as follows.

1. A user navigates to the first page (the checkout page in Figure 6) for an operation and later visits the second page (the order update page in Figure 6).
2. On the first page, there are two REST-based Web services activated to perform two POST actions (creating two types of objects) in a function. On the checkout page in Figure 6, an order object and a payment object are added to the servers.
3. On the second page, one Web service used on the first page is activated to perform a PUT action (updating the saved object). On the order update page in Figure 6, the order object is updated for modifying the gift message.

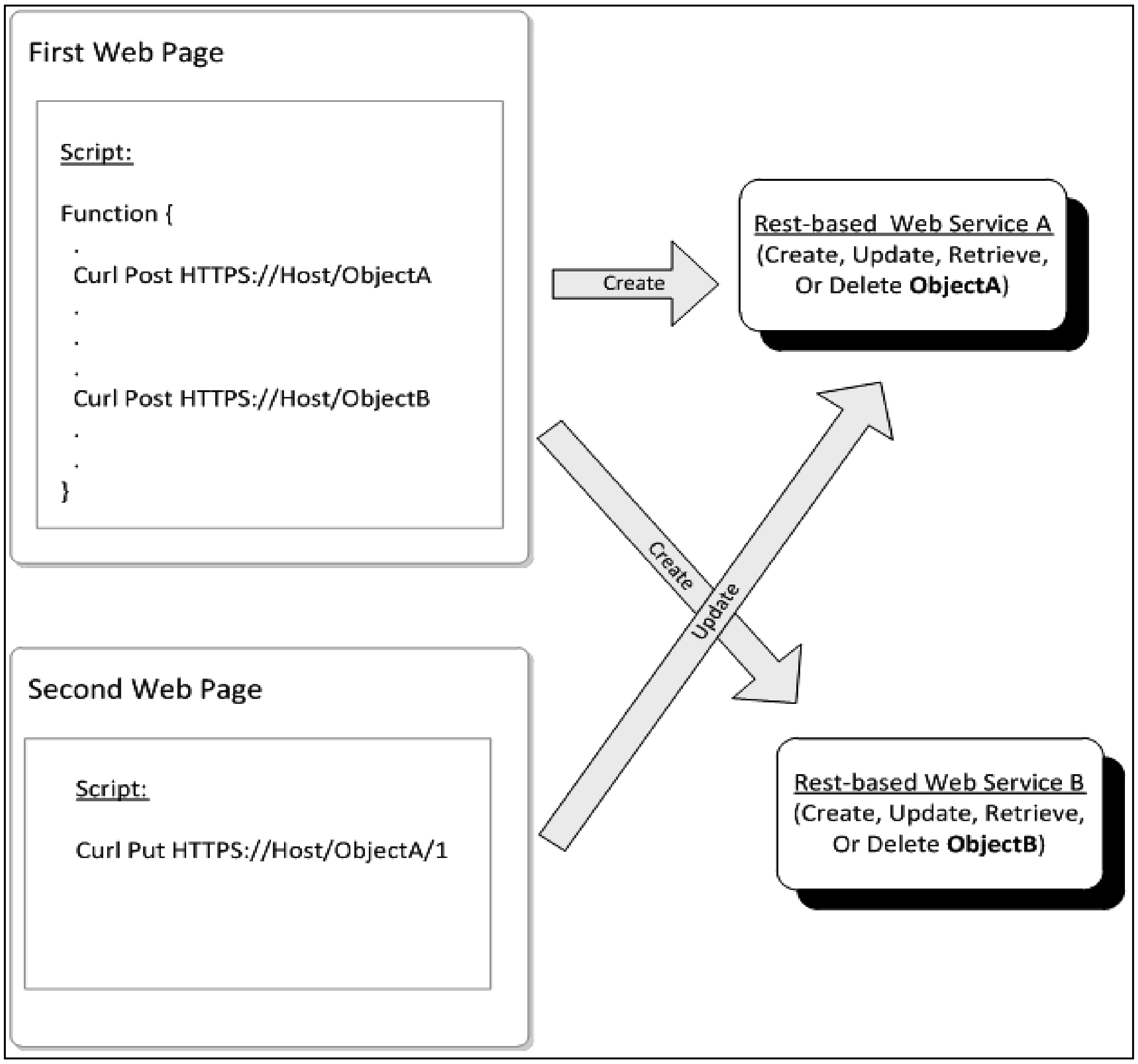

First Web Page
Script:
Function {
Curl Post HTTPS://Host/ObjectA
Curl Post HTTPS://Host/ObjectB
}
Second Web Page
Script:
Curl Put HTTPS://Host/ObjectA/1
Create
Create
Update
Rest-based Web Service A
(Create, Update, Retrieve, Or Delete ObjectA)
Rest-based Web Service B
(Create, Update, Retrieve, Or Delete ObjectB)

Figure 7: JSON Injection—A Vulnerable Pattern of Rest-based Web Service Execution.

The static analysis tool should raise a warning if a HTTP protocol is used for the data transfers in the above page scripts.

CONCLUSIONS

We study the application-level security attacks in an enterprise information system, where REST-based Web services are connected using an MVC framework and utilize JSON as a data-exchange format. While JSON allows developers to easily construct and quickly parse messages transferred through REST Web services, it also leaves these Web services open to potentially serious attacks. An example of a JSON injection input attack was used to illustrate such a threat. The paper investigated this security issue caused by REST-based Web services and presented a couple of possible alternative to defend against such attacks in addition to techniques used to secure the operating systems of Websites. Input validation is quick and easy to filter some suspicious inputs. In spite of being an expensive protection method, message encryption by HTTPS ultimately ensures the confidentiality of critical data. To ensure that programmers activate the HTTPS-based data transfer appropriately, application development teams may adopt static analysis to detect and correct software implementation flaws before software releases.

REFERENCES

Bishop, M., & Dilger, M. (1996). Checking for race conditions in file accesses. *Computing Systems*, *9*(2), 131-152. Retrieved from http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/1996-compsys.pdf

Brown, A. W., Delbaere, M., Eeles, P., Johnston, S., & Weaver, R. (2005). Realizing service-oriented solutions with the IBM Rational Software Development Platform. *IBM Systems Journal*, *44*(4), 727-752.

Ceri, S., Fraternali, P., & Bongio, A. (2000). Web modeling language (WebML): A modeling language for designing Web sites. *Computer Networks*, *33*(1), 137-157. Retrieved from http://58.59.135.118:8081/BOOKS%5C026%5C21%5CHXYWPJH144310.pdf

Chess, B., & McGraw, G. (2004). Static analysis for security. *Security & Privacy, IEEE*, *2*(6), 76-79. doi: 10.1109/MSP.2004.111

Comerford, C., & Soderling, P. (2010, February 24). Why REST security doesn't exist? *Computerworld*, Retrieved from http://www.computerworld.com/s/article/9161699/Why_REST_security_doesn_t_exist

Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *Software, IEEE*, *19*(1), 42-51. Retrieved from http://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/splint.pdf

Jaamour, R. (2005). Securing web services. *Information Systems Security*, *14*(4), 36-44.

Jain, A. K., & Shanbhag, D. (2012). Addressing security and privacy risks in mobile applications. *IT Professional*, *14*(5), 28-33. doi: ieeecomputersociety.org/10.1109/MITP.2012.72

Kennedy, S., Stewart, R., Jacob, P., & Molloy, O. (2011). StoRHm: A protocol adapter for mapping SOAP based web services to RESTful HTTP format. *Electronic Commerce Research*, *11*(3), 245-269. doi: 10.1007/s10660-011-9075-3

Kundu, A., Sural, S., & Majumdar, A. K. (2010). Database intrusion detection using sequence alignment. *International Journal of Information Security*, *9*(3), 179-191.

Lim, B. B. L., Sun, Y., & Vila, J. (2004). Incorporating WS-security into a web services-based portal. *Information Management & Computer Security*, *12*(3), 206-217. doi: 10.1108/09685220410542570

Makino, S., Tamura, K., Imamura, T., & Nakamura, Y. (2004). Implementation and performance of WS-security. *International Journal of Web Services Research*, *1*(1), 58-72. doi: 10.4018/jwsr.2004010104

Manago, W. (2011). Protect, maintain information integrity to reduce business risk. *Information Management*, *45*(3), 36-41.

Margulius, D. L. (2006, July 13). Banking on SOA. *InfoWorld*. Retrieved from http://www.infoworld.com/t/architecture/banking-soa-389

Mitchell, R. L. (2006, January 30). Morphing the mainframe. *Computerworld*, *30*(5), 29-31.

Ortega, R. (2007). Defending the corporate crown jewels from the dangers that lurk within: Effective internal network security focuses on behavior. *Information Systems Security*, *16*(1), 54-60.

Pistoia, M., & Erlingsson, U. (2008). Programming languages and program analysis for security: A three-year retrospective. *ACM SIGPLAN Notices*, *43*(12), 32-39. doi: 10.1145/1513443.1513449

Severance, C. (2012). Discovering JavaScript object notation. *Computer*, *45*(4), 6-8. doi: ieeecomputersociety.org/10.1109/MC.2012.132

Shar, L. K., & Tan, H. B. K. (2012). Defending against cross-site scripting attacks. *Computer*, *45*(3), 55-62. doi: ieeecomputersociety.org/10.1109/MC.2011.261

Shehab, M., & Marouf, S. (2012). Recommendation models for open authorization. *IEEE Transactions on Dependable and Secure Computing*, *9*(4), 583-596. doi: 10.1109/TDSC.2012.34

Su, C. -J., & Chiang, C. -Y. (2012). Enabling successful Collaboration 2.0: A REST-based web service and Web 2.0 technology oriented information platform for collaborative product development. *Computers in Industry*, *63*, 948-959.

Valli, C. (2006). SQL Injection: Threats to medical systems; Issues and countermeasures. Paper presented at *The 2006 World Congress in Computer Science, Computer Engineering, and Applied Computing*. Las Vegas, Nevada, USA.

Wasson, M. (2012). *Working with SSL in Web API*. Retrieved from http://www.asp.net/web-api/overview/security/working-with-ssl-in-web-api

Werts, J. D., Mikhailova, E. A., Post, C. J., & Sharp, J. L. (2012). An Integrated WebGIS framework for volunteered geographic information and social media in soil and water conservation. *Environmental Management*, *49*, 816–832.

Ying, M., & Miller, J. (2013). Refactoring legacy AJAX applications to improve the efficiency of the data exchange component. *Journal of Systems and Software*, *86*(1), 72-88. doi: 10.1016/j.jss.2012.07.019