ISSN No. 0976-5697

Volume 2, No. 5, Sept-Oct 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Security Coding Technique for Web Applications

Shylaja Akinapally*
M.Tech(CSE)
Sree Chaitanya College Of Engineering
Karimnagar,A.P.India
shylaja_akinapally@rediffmail.com

K.Srinivas
Assoc.Prof,CSE Dept
Sree Chaitanya College Of Engineering
Karimnagar,A.P,India.
kaparthisrini@yahoo.com

T.P.Shekhar
Assoc.Prof,IT Dept
Sree Chaitanya College Of Engineering
Karimnagar,A.P,India
tpshekhar@gmail.com

A.Sanjeeva Raju
Assoc.Prof,CSE Dept
Kamala Institute of Technology&Science,Singapur
Huzurabad,Karimnagar,A.P,India.
sanjeevaraju@rediffmail.com

***Abstract*----**We propose a method that provides information-theoretic security for client-server communications. An appropriate encoding scheme based on wiretap codes is used to show how a client -server architecture under active attacks can be modeled as a binary-erasure wiretap channel. The secrecy capacity of the equivalent wiretap channel is used as a metric to optimize the architecture and limit the impact of the attacks. We also provide a method to design attack-resistant client-server architectures that are resilient and secure using wiretap codes. Specifically, the objective is not only to ensure reliable communication between client and servers in the presence of disrupted nodes, but also to guarantee that a malicious attacker hacking the packet information at compromised nodes is unable to retrieve the content of the message being exchanged. In principle, standard encryption techniques could be implemented to ensure secure communication between client and servers; however, instead of using traditional cryptographic tools to encrypt information contained in the packet, the proposed approach exploits the fact that the attacker only gets parts of the packets sent by the client. we define wiretap model as a java web application security framework in order to solve web application vulnerabilities. Wiretap model extends web application's behavior by adding security functionalities maintaining the API and the framework specification. The security functionalities include Integrity, Editable data validation, Confidentiality, Anti-CSRF token.

Keywords- Client-server architecture ,Cross-site scripting, Denial of service, Host compromise attacks, Distributed DoS attack, , network security, parameter tampering, secrecy capacity, SQL Injection , vulnerabilities., wiretap channel

I. INTRODUCTION

Nowadays, web application security is one of the most important issues in the information system development process. According to Gartner the 75% of the attacks performed nowadays are aimed to web applications, because operative system security and net level security have increased considerably. As a result, it is considered that the 95% of the web applications are vulnerable to a certain type of attack. Communication over large networks is often impaired by malicious attacks that aim at disrupting packet traffic. Among the many attacks that infect networks, the most damaging ones are probably denial of service (DoS) and host compromise attacks. In a DoS attack, an attacker tries to direct a large amount of bogus traffic to a susceptible node, with the intention of consuming a large amount of bandwidth and rendering the node unable to service legitimate traffic, whereas in a host compromise attack, an attacker attempts to gain control of a node by exploiting its vulnerabilities. In a more harmful manner, host compromise and DoS attacks can be combined to cause a distributed DoS attack (DDoS), where attackers compromise nodes and use them to launch DoS attacks on a large scale. The frequency and magnitude of DoS attacks have been steadily increasing for the last couple of years [1]. For instance, there has been a significant number of DoS attacks on popular e-commerce websites and governmental websites in 2000 and 2001, and more recently, these attacks have targeted the root domain name servers (DNSs) and the DNS backbone network. Most of the earlier research for countering these attacks has focused on designing schemes capable of detecting attacks and recovering from these attacks using detection mechanisms, such as filtering [2]. More recently, schemes designed to resist attacks have also been proposed, based, for instance, on specialized overlay nodes that have capabilities to resist and survive attacks [3], [4]. Despite all of these protective measures, networks inevitably possess vulnerabilities that attackers may exploit to launch successful attacks; therefore, designing network architectures and additional schemes capable of mitigating the impact of unavoidable attacks has become a crucial issue. In this paper, we consider the effect of these attacks upon the design on resilient and secure client-server architectures.

In the following chart we can see the list of the most important vulnerabilities published by OWASP (Open Web Application Security Project).

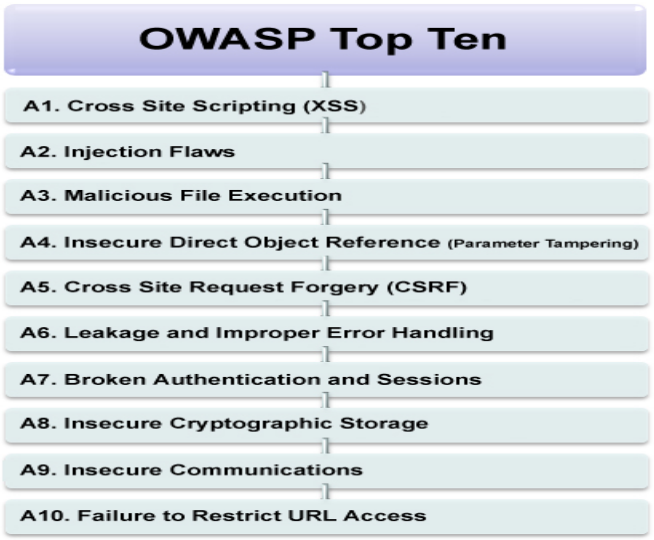

Figure. 1. Vulnerabilities

A. Parameter Tampering:

Parameter tampering is a type of attack based on the modification of the data sent by the server in the client side. The process of data modification is very simple for the user. When a user sends a HTTP request (GET or POST), the received HTML page may contain hidden values, which cannot be seen by the browser but are sent to the server when a submit of the page is committed. Also, when the values of a form are "pre-selected" (drop-down lists, radio buttons, etc.) these values can be manipulated by the user and thus the user can send an HTTP request containing the parameter values he wants.

Example: We have a web application of a bank, where its clients can check their accounts information by typing this URL (XX= *account number*):

```
http://www.mybank.com?account=XX
```

When a client logs in, the application creates a link of this type for each account of this client. So, by clicking in the links, the client can only access to its accounts. However, it would be very easy for this user to access another user account, by typing directly in a browser the bank URL with the desired account number.

For this reason the application (server side) must verify that the user has access to the account he asks for. The same occurs with the rest of non editable html elements that exist in web applications, such as, selection able lists, hidden fields, checkboxes, radio buttons, destiny pages, etc. This vulnerability is based on the lack of any verification in the server side about the created data and it must be kept in mind by the programmers when they are developing a new web application.

Despite being a link the modified element in this example, we must not forget that it is possible to modify any type of element in a web page (selects, hidden fields, radio buttons…). This vulnerability does not only affect to GET requests (links) because POST request (forms) can also be modified using appropriate audit tools, which are very easy to use by anyone who knows how to use a web browser.

B. SQL-Injection:

In this case the problem is based in a bad programming of the data access layer. A SQL-Injection attack consists of insertion or injection of a sql query via the input data from client to application. A successful SQL-Injection exploit can read sensitive data from the database, modify database data (insert/update/delete) , execute administrative operations on the database.

Example: We have a web page that requires user identification. The user must fill in a form with its username and password. This information is sent to the server to check if it is correct.

```
user = john
password = mypassword
```

```
public void doPost(HttpServletRequest request, …) {

      String user = request.getParameter("user");
      String password = request.getParameter("password");

      String sql = "select * from user where username='" + user + "' and
                  password ='" + password + "'";
}
```

```
select * from user where username = 'john' and password = 'mypassword'
```

As we can see in the example, the executed SQL is formed by concatenating directly the values typed by the user. In a normal request where the expected values are sent the SQL works correctly. But we can have a security problem if the sent values are the following ones:

```
user = john
password = mypassword' or '1'='1
```

```
public void doPost(HttpServletRequest request, …) {

      String user = request.getParameter("user");
      String password = request.getParameter("password");

      String sql = "select * from user where username='" + user + "' and
                  password ='" + password + "'";
}
```

```
select * from user where username = 'john' and password = 'mypassword' or '1'='1'
```

In this case, the generated SQL returns all the users of the table, without having typed any valid combination of username and password. As a result, if the program doesn't control the number of returned results, it might gain access to the private zone of the application without having permission for that. The consequences of the exploitation of this vulnerability can be mitigated by limiting the database permissions of the user used by the application. For example, if the application user can delete rows in the table the consequences can be very severe.

C. Cross-Site Scripting (XSS):

This attack technique is based in the injection of code (java script or html) in the pages visualized by the application user.

Example: We have a web page where we can type a text, as is shown in the image below:

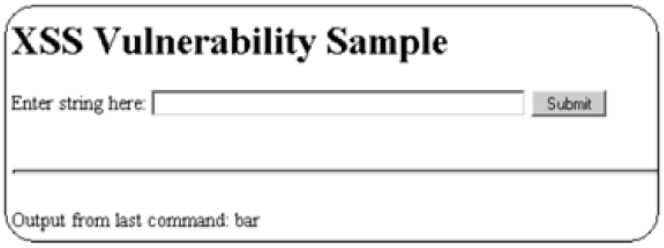

Figure.2. XSS Vulnerability Example.

The html code of the page is:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title>XSS Vulnerability Sample</title></head>
<body>
<h1>XSS Vulnerability Sample</h1>

<form method="GET" action="XSS.jsp">
     Enter string here:
     <input type="text" name="userInput" size=50/>
     <input type="submit" value="Submit" />
</form>
<br><hr><br>
Output from last command: <%= request.getParameter("userInput")%>
</body>
</html>
```

Typing the following text in the textbox:

```
<script>
     alert("If you see this you have a potential XSS vulnerability!");
</script>
```

This is the result:

Figure.3. XSS Vulnerability Example Result.

There is a large variety of attacks to exploit this vulnerability. A well known attack is a massive email sending, attaching a trusted URL (in this example, happy banking) where the final result is the execution of a JavaScript function that can redirect us to another website (a fake website which apparently is the same as original) or can obtain the cookies of our browser and send them to the attacker.

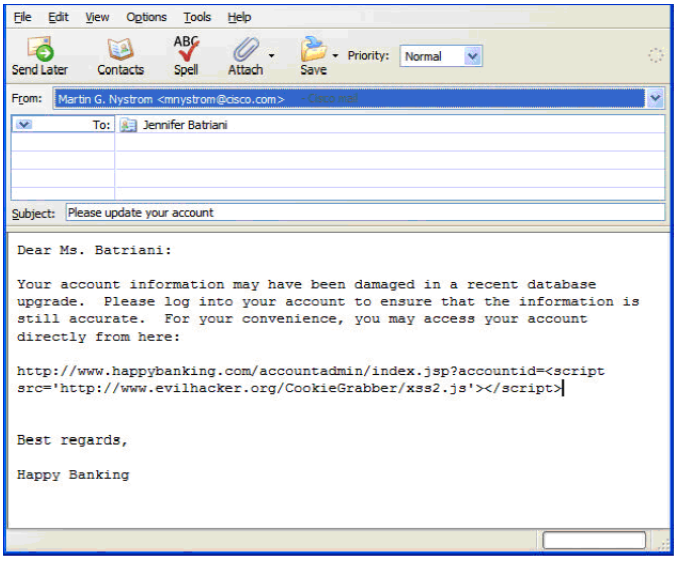

Figure.4. XSS Mail Attack

The rob of cookies can give the attacker access to the web applications where the user is authenticated in that moment (online bank, personal email account, etc.). This is because most of the web applications use cookies to maintain sessions. This vulnerability (XSS) can be solved using generic validation policies (where certain characters are not allowed) or using libraries like Struts which avoids this kind of problems.

D. Cross-Site Request Forgery (CSRF):

Cross-site request forgery, also known as one click attack or session riding and abbreviated as CSRF (Sea-Surf) or XSRF, is a type of malicious exploit of websites. Although this type of attack has similarities to cross-site scripting (XSS), cross-site scripting requires the attacker to inject unauthorized code into a website, while cross-site request forgery merely transmits unauthorized commands from a user the website trusts. 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 authenticated.

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.,

```
<img
src="http://bank.example/withdraw?account=bob&amp;amount=1000000&amp;for=mallor
">
```

If Bob's bank keeps his authentication information in a cookie, and if the cookie hasn't expired, then Bob's browser's attempt to load the image will submit the withdrawal form with his cookie, thus authorizing a transaction without Bob's approval. A cross-site request forgery is a confused deputy attack against a Web browser. The deputy in the bank example is Bob's Web browser which is confused into misusing Bob's authority at Mallory's direction.

The following characteristics are common to CSRF:

a. Involve sites that rely on a user's identity
b. Exploit the site's trust in that identity
c. Trick the user's browser into sending HTTP requests to a target site
d. Involve HTTP requests that have side effects

II. STATE OF ART

All the vulnerabilities presented before can be solved through a proper input validation. There are solutions for this but most of them are custom solutions and developers have to create a new solution for each use case. Also we must add that it's highly probable that developers forget a validation in some points of the web application. In order to solve this problem there are some global solutions. Web application framework validators can be useful to solve problems like *SQL Injection* or *XSS* but it's limited to type validation. We can't solve *parameter tampering* through Struts' validator. With these validators we can assure that a parameter is an integer but we can't know if the value it's the same that the server sent to the client. In other words, we can't assure server data integrity. Avoiding this vulnerability manually implies a great development effort and it is likely to fail in some pages because it is very difficult to test the correct programming of each page.

III. WIRETAP MODEL

A. Introduction:

In order to solve web application vulnerabilities a WIRETAP MODEL is created. We can briefly define WIRETAP MODEL as a Java Web Application Security Framework. WIRETAP MODEL extends web applications' behavior by adding Security functionalities, maintaining the API and the framework specification. This implies that we can use WIRETAP MODEL in applications developed in Struts 1.x, Struts 2.x, Spring MVC or/and JSTL in a transparent way to the programmer and without adding any complexity to the application development.

B. The Security Functionalities Added to the Web Applications :

***a. Integrity*:** WIRETAP MODEL guarantees integrity (no data modification) of all the data generated by the server which should not be modified by the client (links, hidden fields, combo values, radio buttons, destiny pages, cookies, headers, etc.). Thanks to this property we avoid all the vulnerabilities based on the *parameter tampering*.

***b. Editable data validation*:** WIRETAP MODEL eliminates to a large extent the risk originated by attacks of type *Cross-site scripting (XSS)* and *SQL Injection* using generic validations of the editable data (text and text area). As there isn't any base in editable data to validate the information, the user will have to configurate generic validations through rules in XML format, reducing or eliminating the risk against attacks based on the defined restrictions. Unlike the traditional solution where validations are applied to each field through the Commons validator, and where the probability of a human error is very high, WIRETAP MODEL allows to apply generic rules that avoid to a large extent the risk within these data types. Anyway, it is advisable to use existing solutions such as the Struts' validator and Struts' tag libraries to avoid *Cross-site scripting (XSS)* attacks and to use prepared statements to avoid *SQL injection* in the data access layer. The responsibility of showing error messages on the user screen, if the WIRETAP MODEL validator detects not allowed values in editable fields, is delegated to the errors handler and this handler will show them in the input form.

c. Confidentiality: WIRETAP MODEL guarantees the confidentiality of the data as well. Usually lots of the data sent to the client has key information for the attackers such as database registry identifiers, column or table names, web directories, etc. All these values are hidden by WIRETAP MODEL to void a malicious use of them. For example a link of this type, http://www.host.com?data1=12&data2=24 is replaced by http://www.host.com?data1=0&data2=1, guaranteeing confidentiality of the values representing database identifiers.

d. Anti-CSRF token: Random string called a token is placed in each form and link of the HTML response, ensuring that this value will be submitted with the next request. This random string provides protection because not only does the compromised site need to know the URL of the target site and a valid request format for the target site, it also must know the random string which changes for each visited page.

Therefore, WIRETAP MODEL helps to eliminate most of the web vulnerabilities based on non editable data and it can also avoid vulnerabilities related with editable data through generic validations, which is easier to apply than traditional input validation with the Commons Validator.

In addition to that, WIRETAP MODEL hides all critical information to the client to avoid a malicious use of them

C. Base Concepts:

Before detailing the way WIRETAP MODEL guarantees data integrity and confidentiality it is necessary to explain some base concepts.

a. ***State:*** For WIRETAP MODEL a state represents all the data that composes a possible request to a web application, that is, the parameters of a request, its values and its types and the destiny or page request. We may have more than one state (possible request) for a page which represents the links and forms existing in the page. When a page (JSP) is processed in the server, WIRETAP MODEL generates an object of type state for each existing link o form in the page (JSP).Generated state can be stored in two locations:

b. ***Server:*** States are stored inside de session (Http Session) of the user.

c. ***Client:*** State objects are sent to the client as parameters. For each possible request (link or form) an object that represents the state of the request is added. These states make it possible the later verification of the requests sent by the clients, comparing the data sent by the client with the state.

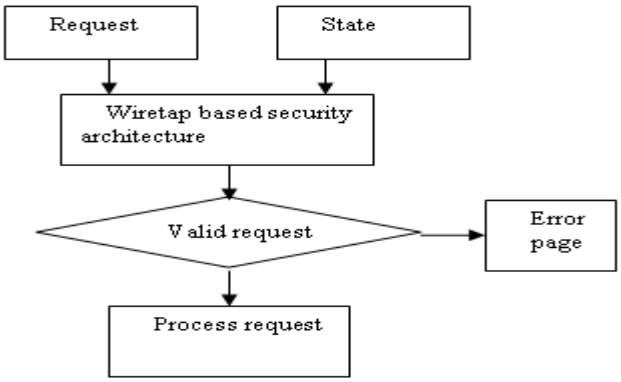

Figure.5. Validation process

D. Architecture:

WIRETAP MODEL has two main modules:

a. ***Tag Library*****:** Tag Library is responsible for modifying the html content sent to the client that then will be checked by the security filter.

b. ***Security Filter*****:** It validates the editable and non editable information of the requests, using the generic validations defined by the user for editable data and the state received in the requests for the non editable information.

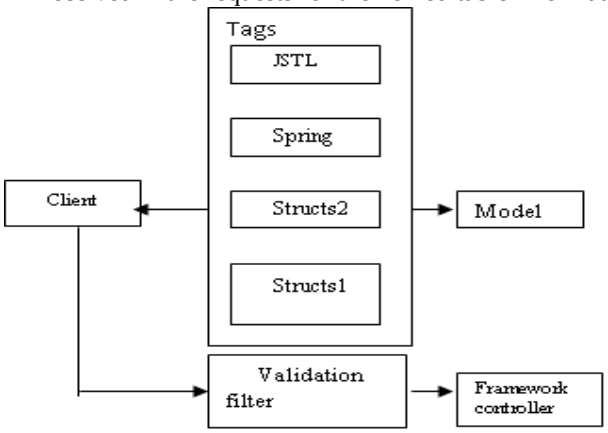

Figure.6. Architecture

IV. OPERATION STRATEGY

Having the same objectives, WIRETAP MODEL has different operation strategies: Let's see the html code generated by WIRETAP MODEL using different strategies and configurations as well as the steps of the validation process. Suppose that we have a page that generates the following html code, where shaded text represents non editable data that we must protect.

```
<html>
<body>
<a href=/struts-examples/action1.do?data=22>LinkRequest</a>

<form method="post" action="/struts-examples/processSimple.do">
     <input type="text" name="name" value=""/>

     <input type="password" name="secret" value="" />
     <select name="color">
          <option value="10">Red</option>
          <option value="11">Green</option>
          <option value="21">Blue</option>
     </select>

     <input type="radio" name="rating" value="10" />Actually, I hate
it.<br />
     <input type="radio" name="rating" value="20" />Not so much.<br />
     <input type="radio" name="rating" value="22" />I'm indifferent<br
/>
     <textarea name="message" cols="40" rows="6"></textarea>
     <input type="hidden" name="hidden" value="15" />
     <input type="submit" value="Submit" />
</form>
</body>
</html>
```

A. Cipher Strategy:

The state is sent to the client as a hidden field or a parameter if it is a link. In order to guarantee integrity, the state is ciphered using a symmetrical algorithm. In order to guarantee confidentiality, non editable data is replaced by relative values.

a. Response Generation:

First of all WIRETAP MODEL gathers all the request data and it generates an object of type org .WIRETAP .state. IState for each request of the page (forms + links). This State object is what the client receives as a serialized object. Then, WIRETAP MODEL replaces non editable real values by relative values. For instance, if we have a selection list with the following values: 150, 133, 22 they are replaced by these: 0, 1, 2. This way WIRETAP MODEL guarantees confidentiality of non editable data. Once IState object is created, it will be sent to the client as a hidden field for the forms and as an extra parameter for the links.

These are the steps to get the value of this parameter:

i. An array of bytes of the IState object is obtained (the object must be serializable).
ii. It is compressed.
iii. It is ciphered

iv. It is coded to Base64.

The result of a page using the Cipher strategy and which has activated confidentiality flag
will be like this:

```
<html>
<body>
<a href=/struts-examples/action1.do?data=0&_WIRETAP MODEL_STATE=6347dfhdfd84r73e9483494734837487> LinkRequest</a>
<form method=”post “ action=”/struts-examples/processSimple.do”>
<input type=”text” name=”name” value=””/>
<input type=”password” name=”secret” value=””/>
<select name=”color”>
<option value=”0”>Red</option>
<option value=”1”>Green</option>
<option value=”2”>Blue</option>
</select>
<input type=”radio” name=”rating” value=”0”>Actually, I hate it<br/>
<input type=”radio” name=”rating “ value=”1”>Not so much<br/>
<input type=”radio” name=”rating ” value=”2”> I am indifferent<br/>
<textarea name=”message” cols=”40” rows=”6”/>
<input type=”hidden” name=”hidden” value=”0”/>
<input type=”hidden” name=”WIRETAP MODEL_STATE”
value=”jkfhdfhgdf948dkfhghfdkhfffjfdf”/>
<input type=”submit” value=”submit”/>
</form>
</body>
</html>
```

b. Validation:

The first step in the validation process is to decrypt the value of the _WIRETAP MODEL_STATE_ parameter, which has the state of the request. If there is no error decrypting the state, it means that the value hasn't been modified and so we must continue with the validation process. The next step is to decompress the parameter value and to create a new IState object from the obtained bytes. Once we have the IState object we are ready to validate the client request. WIRETAP MODEL will check each of the parameters of the request and all the values of each parameter.

```
While (parameters) {
      While (values)
      {
            DataValidator.validate(value);
      }
}
```

First of all, WIRETAP MODEL verifies that the parameter value is between the possible relative values for the parameter. If it is correct WIRETAP MODEL returns the real value of the option selected by the client. For example, if the relative value of the account parameter is 0, WIRETAP MODEL replaces it by the value in the 0 position on the list of values for this parameter. If the request is correct it is redirected to the Struts controller to generate the corresponding page. Otherwise, if a security error is detected the user is redirected to an error page and the incident is logged on a file.

B. Hash Strategy:

The state is coded in Base64 and sent to the client as a hidden field or as a parameter if it is a link. In order to guarantee integrity, before sending the state to the client a hash of the state is generated and it is stored in the user session. Later, this will be use to check that the value hasn't been modified. The main difference between this strategy and Cipher strategy is that here the state integrity is guaranteeing using a hash, instead of ciphering the state object. It is worth mentioning that in this case data confidentiality can't be guaranteed, as the data is not ciphered. On the other hand, no real values are sent inside each component in order to make it more difficult to know the real values.

a. Response Generation:

Visually the result of a page using this strategy is the same as the previous one

```
<html>
<body>
<a href=/struts-examples/action1.do?data=0&_WIRETAP MODEL_STATE=wJTAwJTAwbvAlQzFKJUMzJTQwJTE
>
LinkRequest</a>
<form method=”post “ action=”/struts-examples/processSimple.do”>
<input type=”text” name=”name” value=””/>
<input type=”password” name=”secret” value=””/>
<select name=”color”>
<option value=”0”>Red</option>
<option value=”1”>Green</option>
<option value=”2”>Blue</option>
</select>
<input type=”radio” name=”rating” value=”0”>Actually, I hate it<br/>
<input type=”radio” name=”rating“ value=”1”>Not so much<br/>
<input type=”radio” name=”rating” value=”2”> I am indifferent<br/>
<textarea name=”message” cols=”40” rows=”3”/>
<input type=”hidden” name=”hidden” value=”0”/>
<input type=”hidden” name=”_WIRETAP MODEL_STATE” value=”lRMwUHDSFRwGFDgew CX fj”/>
<input type=”submit” value=”submit”/>
</form>
</body>
</html>
```

b. Validation:

The only difference with the Cipher strategy is that the decrypting process is replaced by the integrity verification using the hash. In order to check the integrity WIRETAP MODEL calculates the hash of the value that represents the

state and it is compared with the one stored in session. From this point, the request verification is exactly the same as on the previous strategy.

C. Memory Strategy:

The state of each request is stored in the user session, being this the main difference with the other two strategies. To be able to associate user requests with the state stored in the session, an extra parameter (_WIRETAP MODEL_STATE_) is added to each request. This parameter contains the identifier that makes possible to get the state from session.. In order to guarantee confidentiality, non editable data are replaced by relative values.

a. Response Generation:

The difference with the other two strategies is that here the state is not sent to the client. Only the request identifier is sent in order to be able to recover the request state later.

```
<html><body>
 <a href=/struts-examples/action1.do?data=0&_WIRETAP
MODEL_STATE=0-1-5E26F18AD9E>
LinkRequest</a>
<form method=”post “ action=”/struts-
examples/processSimple.do”>
<input type=”text” name=”name” value=””/>
<input type=”password” name=”secret” value=””/>
<select name=”color”>
<option value=”0”>Red</option>
<option value=”1”>Green</option>
<option value=”2”>Blue</option>
</select>
<input type=”radio” name=”rating” value=”0”>Actually, I
hate it<br/>
<input type=”radio” name=”rating “ value=”1”>Not so
much<br/>
<input type=”radio” name=”rating ” value=”2”> I am
indifferent<br/>
<textarea name=”message” cols=”40” rows=”3”/>
<input type=”hidden” name=”hidden” value=”0”/>
<input type=”hidden” name=”_WIRETAP
MODEL_STATE” value=”0-2-5E26F18AD9E”/>
<input type=”submit” value=”submit”/>
</form>
</body></html>
```

As we can see there are not visual differences in the generated html code between state in client or state in server versions. The only difference is the length of the parameter that represents the state (_WIRETAP MODEL_STATE_), which is much shorter in this case because it only contains the request identifier.

b. Validation:

Before initializing validation, WIRETAP MODEL obtains the request identifier and thus the object of type org. WIRETAP MODEL. State. IState is obtained from user session. From this point, the validation process is exactly the same as the previous strategies.

V. CONCLUSION

We have investigated a method that provides information-theoretic security for client-server architectures. By introducing an appropriate encoding scheme we showed how client-server architecture under active attacks can be modeled as a binary erasure wiretap channel, which motivates the use of wiretap coding before packet transmission. Our analysis supports the idea that wiretap channel models can be used beyond standard communication problems, even in situations where the presence of active attackers is assumed. Most of the web application vulnerabilities are solved using this wiretap security model.

VI. ACKNOWLEDGMENT

The Authors Would like to thank the Reviewers for their usful comments that Greatly Improved the Presentation and clarity of this paper.

VII. REFERENCES

[1] D. Moore, G. Voelker, and S. Savage**,** “Inferring internet denial-of-service activity,” in *Proc. USENIX Security Symp.*,Washigton, D.C., Aug.2001, pp. 9–22.

[2] P. Ferguson and D. Senie, “Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing,” *RFC 2827*, May 2000.

[3] S. Kandula, D. Katabi, M. Jacob, and A. Berger, “Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds,” presented at the 2nd Symp. Networked Systems Design and Implementation, Boston,MA, May 2005.

[4] T. Bu, S. Norden, and T.Woo**,** “A survivable dos-resistant overlay network,”*Comput. Netw.*, vol. 50, no. 9, pp. 1281–1301, Jun. 2006.

[5] T. Bu, S. Norden, and T. Woo, “Trading resiliency for security: Model and algorithms,” in *Proc. 12th IEEE Int. Conf. Network Protocols*, Berlin, Germany, 2004, pp. 218–227.

[6] R. Narasimha, Z. Chen, and C. Ji, “Topological malware propagation on networks: Spatial dependence and its significance,” *IEEE Trans. Secure Dependable Comput.*, 2007, submitted for publication.

[7] C. E. Shannon**,** “Communication theory of secrecy systems,” *Bell Syst. Tech. J.*, vol. 28, pp. 656–715, 1948.

[8] A. D.Wyner, “The wire-tap channel,” *Bell Syst. Tech. J.*, vol. 54, no. 8, pp. 1355–1367, Oct. 1975.

[9] L. H. Ozarow and A. D.Wyner, “Wire tap channel II,” *AT&T Bell Laboratories Tech. J.*, vol. 63, no. 10, pp. 2135–2157, Dec. 1984.

.