

# How to Counter Control Flow Tampering Attacks

Mohsen Sharifi, Mohammad Zoroufi, Alireza Saberi  
*msharifi@iust.ac.ir, {m\_Zoroufi, a\_Saberi}@comp.iust.ac.ir*  
Computer Engineering Faculty  
Iran University of Science & Technology

## Abstract

Nowadays more and more business activities are operated through web and the web plays a vital role in the interests of both businesses and their shareholders. However, the very good features of web such as its popularity, accessibility and openness, has provided more opportunities for security breaches by malicious users. That is why the rate of successful attacks on web and web applications are increasing. Many approaches have been introduced so far to reduce the rate of successful attacks of many kinds. Any technique that can detect these vulnerabilities and mitigate the security problems of web applications is useful to organizations seeking for more reliability from the security viewpoint. In this paper we first introduce the control flow tampering attack, which is one of the notable attacks against web applications, and present our approach for countering this attack using web application firewall.

## 1. Introduction

Deployment of web applications in Internet has critical prerequisites whose unavailability can cause havoc to the providers of these applications. The noticeable features of web applications that have made them a good candidate for E-Business are that they require no special configuration at the end user side and thus can be easily deployed by a simple browser at the user side, and they can be accessible to users of all kinds through internet. The latter feature has provided the potential for all sorts of security breaches and attacks by malicious users as well. So securing web applications by all means is vital to the success of these applications.

In this paper we present an especial kind of attack targeting most web applications and then propose a technique detecting and countering this attack in web application firewalls.

The rest of paper is organized as follows; section 2 introduces available approaches in detecting application layer vulnerabilities and countering them; introducing a specific kind of attack named *control flow tampering attack* is in section 3; proposing an architecture and implementation in web application firewalls that counter this kind of attack is in section 4; related works in the

web application security, concluding remarks and further works are the last sections of the paper.

## 2. Web application security approaches

There are three main traditional approaches for detecting and countering security breaches to web applications:

1. Auditing the source codes and looking for vulnerabilities inherent in the source of codes and reporting them to software developers for corrective action. This approach is static and useable when the web applications are under development and not deployed and operated in organizations.
2. Auditing and analyzing the behavior of web applications by mimicking the behaviors of hackers in order to detect vulnerabilities; penetration test [4, 8] uses this approach.
3. Using application layer firewalls that work at the HTTP request and response level and sit between the users' browsers and web servers. So by defining policies in these firewalls, the firewalls stop requests and responses that violate these policies [1, 3].

An advantage of web application firewalls in comparison to other approaches is that the policies set in a firewall can be applied to all incoming/outgoing requests/responses. Working on HTTP packet level and having no dependency on specific web application development language is another advantage of using web application firewalls. Its weakness is however that there is no way to discover the vulnerable points that lied at the source of web applications; only renowned vulnerabilities in the body of requests and responses are detected and prevented. This is because traditional web application firewalls are signature based and only vulnerabilities that have a defined signature in the firewall repository are detected and prevented.

## 3. Our proposition

An especial kind of vulnerabilities that causes web applications behave unpredictably is the subject of this research.

In this section we will briefly introduce the vulnerability first, and then propose our under development technique for detecting and stopping it.

### 3.1. Control flow tampering attacks

In most web applications this is important that where we are going to. However the more important is that where we are coming from? In other words the previously requested URL and the newly requested and the order of them are important. This is required in order to investigate the legitimacy of requests initiated by users. For example, in most web applications, the sequence of requests by a user must have been initiated in a predefined order. If the order is tampered intentionally or inadvertently by any legitimate or malicious user, it can lead to security violation.

For example, Shopping Cart applications require a specific navigation sequence for their web pages by visitors or customers using their web sites. Customers add their intended goods into their Shopping Carts in the early web pages. The purchase transaction is committed when the prices of the selected goods are paid. After adding the intended goods into the Cart, if a malicious user can bypass the web pages that authenticate the customer and accept a credit card number, the goods could be bought without payment if developers of the web application have not attended to such cases in the source codes.

### 3.2. Security gateway

In order for a firewall to be able to detect any kind of web application vulnerability some prerequisites are needed. To achieve the ability to detect and counter the *control flow tampering attack* web application firewalls need the followings:

1. Navigation orders or rules among all the pages in the web site are figured out. A *dependency graph* is required for this purpose.
2. The source of request and the requested URL is figured out. Each request received by the server must contain information about where it has come from and which URL is asking for?
3. The origin of the incoming request is looked in the dependency graph and the requested URL is in its neighboring nodes is found.

If the specified request conforms to the specified rules it is accepted, and otherwise the request is rejected and reported as a security breach. These actions can be performed by a security gateway.

The navigation flow between the pages can be defined by security experts who know exactly the order of navigation for web pages.

The definition of navigation rules among the web pages, or the construction of dependency graphs, can be

done in two ways (note that it is assumed that after graph construction, the behavior of firewall is fixed and cannot change):

- Manually by a security expert.
- Automatically at run time by an engine such as a *flow crawler*.

The choice of second alternative in our approach has its own performance costs. This is because all links in a web application are used to construct a complete dependency graph, where only part of these links may be important and useful for security consideration. The cost of looking up a node in the whole graph and validation of each request can be unnecessarily high. A better performance is indeed attainable when human experts manually consider only the security concerned pages in the dependency graphs, leading to smaller graphs with fewer search overheads for nodes and edges.

The proposed architecture has the following major parts more than traditional security gateways:

- A *flow crawler* that crawls exhaustively all the web site.
- *Dependency graph* representing the whole structure of the web site.
- An engine that used to construct the dependency graph.

The *flow crawler* differs from traditional crawlers that crawl only using existing hyperlinks in web pages. Our crawler is a *deep crawler* [8, 9] or *hidden crawler* that performs automatic form filling and submission.

Dependency graph is a data structure dedicated for representing the structure of the web site. This data structure is public to all online users constructed using the *flow crawler* and the engine used for this.

Finally an engine that used for constructing the dependency graph using the *flow crawler*; the task of this engine is limited to the time slice of the life cycle of the security gateway that is going to startup, but no incoming requests and outgoing ones will be allowed up to time that the dependency graph is constructed.

### 3.3. Validation of requests

As noted before, in order to prevent CFTA attacks to a web site, we propose to have the dependency graph of the site. This graph is a directed graph  $G(V, E)$  where  $V$  represents the nodes (i.e. the URL of pages requested by users) of the graph, and  $E$  stands for the edges (i.e. the direction and order of URL requests).

To construct a dependency graph for a web site, a robot called *flow crawler* crawls the site remotely from the welcome page of the site, by scanning the content of the page and extracting all the link(anchors, submit buttons and so on) in that page. This builds the first level of directed graph. Deeper levels of graph are iteratively

built similarly by scanning the content of a lower level node and extracting all the links in that page.

Now how a running security gateway can detect and prevent CFTA attacks using the dependency graph of a site? The gateway keeps a marker for each user who has been authenticated by and passed through the web server. Each user is represented by its *session id* given to it by the web server. The marker shows the whereabouts of the user within the dependency graph of the corresponding site. The marker is pointed to the start node of the graph upon successful sign in of the user. Whenever a request is received by the gateway, it can check the validity of the request by looking at the position of the marker for the user issuing the request. If the requested URL is in the neighboring nodes of the node pointed by the marker, the gateway passes the request to the web server as a legitimate request, and otherwise rejects the request.

It should be pointed out that all web pages that are created dynamically on the fly by web servers are not found out by the crawler in one time crawling. So our proposition may not function properly in real web application firewalls. To overcome this problem we propose that the crawling process of the web application is iterated multiple times so that multiple execution of the web application is available when constructing the dependency graph of the web application.

## 4. Implementation

To evaluate the proposed firewall we are currently developing a java-based tool using the open source *WebScarab* [5] developed by the OWASP. *WebScarab* contains plug-ins that can be added and removed from it. One of the plug-ins is the proxy that gets all incoming requests and outgoing responses and does any analysis on each http request/response.

Requiring a crawler for construction of dependency graph, we are developing a tool based on the *httpunit* [7] which is an open source java based library.

As stated before there are two steps in empowering the traditional web application firewalls in detecting and protecting web applications from this kind of vulnerabilities:

1. Construction of the dependency graph.
2. Tracking all user requests.

Firstly we did this and constructed the dependency graph of web application. The dynamic nature of almost all web applications requires navigation of all possible paths existing in web applications. But this is impossible since existing technologies don't provide such capability.

In order to be able to build the dependency graph that contains probably all the navigation paths in a web application, we investigated some techniques that resemble the graph to ideal graph. These techniques are:

1. Executing the dependency graph construction process multiple times with different test cases.
2. Using a knowledge base to help us in filling the form fields with possible appropriate values [8].

In order for mimicking the behavior of a web browser we used a tool for crawling the whole site. This tool is named *httpunit* [7]. We started a web site crawling from the welcome page and extended the scope of crawling by extracting all the form entries and hyperlinks in the body of pages.

At the first glance it seems that the resulting dependency graph may be fat even for a small web application. But we must consider that this data structure is public and common for all users' session, so only one copy of it will be constructed for ever.

The major part of our duty is the automatic data entry of filling form. To do this we encounter problems such as: we must know the aim of each entry in the body of web forms in order to fill the entry appropriately. We have exploited technique for automatic form submission proposed by Huang et. al. [8], so we use a topic model that is based on this topic model and get the possible values from the Knowledge Manager and submit the forms to the web server to get the response from it.

We use a repository for storing and retrieving all username/passwords existing in the system in order to use them to authenticate to the system.

### 4.1. Evaluation

Although our approach leads to a graph that may be different from the ideal graph, the differences are due to the following:

1. There is no guarantee that all navigation paths in the body of web applications are identified.
2. Filling the form's data entries with all possible values is not feasible to discover all navigation paths.
3. Discovering some navigation paths need some server internal conditions that may not be satisfied.
4. Due to inaccessibility to the source of web applications, figuring out the best values of navigation paths is not possible.

To overcome these, we adopted tricks that mitigate these concerns. Firstly almost at every application iterating a process may lead us in having a complete and sound model. Secondly we tried all the possible values to be submitted to the web server. For example if any form contains a parameter that may have some domain value (for example list of countries stored in a list box, marital status stored in a radio button or etc.) we submitted the http request for each possible case. Because of connection less nature of the web, sessionid is the one way to tracking each request, So after constructing the graph we

associate a unique id for each sessionId in order for storing the current request initiated by the legitimate user. We named this unique id as *marker*, So when a Http request is received by the gateway, the marker is looked up if the new request initiated by the user is one of neighbors of the marker in the graph we call that this request is legitimate or malicious else.

## 5. Related works

Scott and Sharp [1, 3] used a web application firewall that intercepts all the incoming requests and outgoing responses. They believe that there is no way to guarantee that the source code developed by developers is secure because of the lack of security knowledge and experiences of all web application development teams. So by specifying security rules and constraints by a security expert, declaring them in *SPDL*, malicious requests/ violated responses can be detected.

An advantage of this approach in using web application firewall is that the technology in which the web application is developed will be not important and developers can use any technology for development. This is because their firewall works on http packet coming in and going out via firewall. Another capability of this firewall is that it generates script for form field validation both at server side and client side [1, 3].

Jeff Offutt et. al [6] proposed a technique for testing a web application. They categorized all test cases in three main classes, namely value level, parameter level, and control flow level [6]; in the latter we are looking to detect whether the order of visiting pages are altered or not. Their approach is limited to the test of bypassing a web application intentionally hacked by malicious users but ours is preventative.

## 6. Conclusion and future works

In this paper we introduced an important kind of attack that is triggered by hackers to compromise the security of web applications. We showed how we can detect and encounter this attack by simply adding some changes to and extending the traditional security gateways. Although traditional frameworks and web application development environments have guidelines for security concerns, but there is no guarantee that the developers use them. That is why using a web application firewall is justified to intercept all the incoming requests and outgoing responses and filter malicious requests and violated responses.

Although web application firewall can enhance the detection and prevention of application level attacks to web applications, but some of the implications of using these technologies are listed below:

1. Performance issues of this kind of firewall.
2. Signature based nature of firewalls.
3. Not detecting point of vulnerabilities in the source code.

These encourage us to think of other novel approaches that can cover the shortcomings of firewalls listed above and encompass some necessary primitives that obligate the third parties to use them in their organizations.

In many web applications about 80 percent of the pages are created dynamically on the fly and sent to the users' browsers by the web servers/ application servers. So defining the policies between these dynamic pages has its own concerns and this definition must be taken dynamically and changed by the web servers not by the security experts/analysts.

## Acknowledgments

The authors wish to thank to Iran Telecommunication Research Center (ITRC) for their valuable support of this research.

## 7. References

- [1] D. Scott and R. Sharp, "Abstracting Application- level Web Security", In Proceedings of the 11<sup>th</sup> International Conference on the World Wide Web (WWW 2002), May 2002.
- [2] Web Application Security Consortium: Threat Classification, URL:[http://www.webappsec.org/projects/threat/v1/WASC-TC-v1\\_0.pdf](http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.pdf), 2004.
- [3] D. Scott, R. Sharp, "Specifying and Enforcing Application-Level Web Security Policies", IEEE Transactions on Knowledge and data Engineering, August 2003.
- [4] B. Arkin et al., "Software Penetration Testing", IEEE Security & Privacy, 2005.
- [5] Open Web Application Security Project, WebScarab Project, [www.owasp.org](http://www.owasp.org), 2006.
- [6] J. Offutt et. al. "Bypass Testing of Web Applications", IEEE International Symposium on Software Reliability Engineering, November 2004.
- [7] Httpunit, URL: <http://httpunit.sourceforge.net>.
- [8] Y. Huang et. al., "Web Application Security Assessment by Fault Injection and Behavior Monitoring", ACM, In Proceedings of 12th international conference of World Wide Web Budapest, Hungary, May 2003.
- [9] Bergman, M. K. "The Deep Web: Surfacing Hidden Value", Deep Content Whitepaper, 2001.