******

#### CS 5434 Defending Computer Networks

#### Web Exploit Scanner

#### Name: Prashanth Basappa Net ID: pb476

**Table of Contents:**

1. Introduction………………………………………………………………………………3
   1. Brief Overview
   2. HTTP
   3. Structure of this Document
2. Architecture……………………………………………………………………………….5
   1. Socket Programming
   2. HTTP Proxy
   3. VM ‘honeypot’ browser
   4. File cache system
   5. Malicious checking
   6. Modules
3. Testing……………………………………………………………………………………10
   1. Proxying a client request
   2. Virtual Machine setup
   3. Relaying the response back from the VM
   4. Discriminate malicious from non-malicious domains
4. Conclusions and Future Work………………………………..…………………………13
5. References………………………………………………………………………………..13
6. Screenshots ……………………………………………………………………………14

## **Introduction**

This report describes the architecture of a web exploit scanner implemented as part of CS 5434 Defending Computer Networks project in Cornell University (2014-15). The function of a web exploit scanner is to detect a list of malicious domains on a separate independent machine and relay it back to the client via a HTTP proxy.

## **1.1** **Brief Overview**

In this project, a HTTP proxy was implemented which can interact with a virtual machine and replay contents from a server(web page) and detect if it is malicious or not. This web exploit scanner can detect malicious websites/domains and report it back to the client browser. The HTTP proxy will handle connections to the client (browser), server (web page) and virtual machine via sockets. It keeps track of the web pages parsed in a local file cache (~64KB).

**1.2 The Hypertext Transfer Protocol**

The Hypertext Transfer Protocol or (HTTP) is the protocol used for communication on web. That is, it is the protocol which defines how our web browser requests resources from a web server and how the server responds.

HTTP communications happen in the form of transactions, a transaction consists of a client sending a request to a server and then reading the response. Request and response messages share a common basic format:

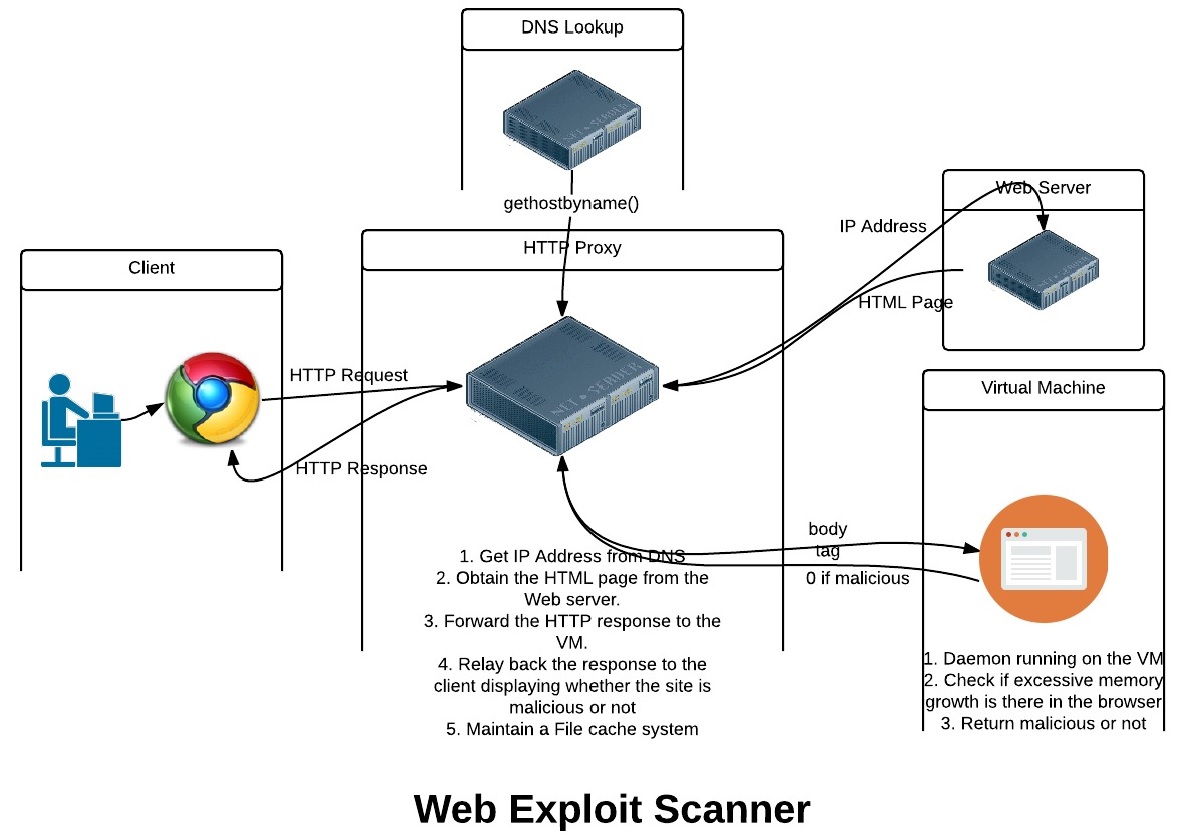
* An initial line (a request or response line, as defined below)
* Zero or more header lines
* A blank line (CRLF)
* An optional message body.

We use this information to parse the request and response messages to extract the necessary information for our proxy.

## **1.3 Structure of this Document**

The remainder of this document is arranged as follows: section two describes the software architecture and the various design decisions made. It describes in detail the working of the web exploit scanner. Section three describes the tests that have been/would be performed to test the correctness and scalability of this scanner. Section four briefly describes how this exploit scanner can be expanded to add more features and further improvements that can be made to it. Section five are the references. Section six contains some screenshots.

## **Architecture:**



### **2.1 Socket Programming:**

### Listening

When the proxy starts, a socket connection was established to listen for incoming client connections on a specified port (say 8080). Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request. An invalid request from the client is answered with an appropriate error code.

### Parsing the URL

Once the proxy sees a valid HTTP request, requested URL is parsed. The proxy needs at most three pieces of information: the requested host and port, and the requested path.

### Getting Data from the Remote Server

Once the proxy has parsed the URL, it can make a connection to the requested host (using the appropriate remote port) and send a HTTP request for the appropriate file. The proxy then sends the HTTP request that it received from the client to the remote server.

### Sending data to the VM

Once the proxy receives the data from the server, it stores it in the buffer. An independent port is created and a new connection is created to the VM daemon that will be listening just as the proxy. Once all the data comes in the VM, file operations are performed (html file) to the incoming buffer data and stored as an html file in the local disk (say temp.html).

* Returning decision whether the website is malicious or not

Use the same socket from where the buffer data was received to send back a decision whether the website is malicious or not. This will be a simple integer data type.

### Returning Data to the Client

After the response from the virtual machine is received, the proxy should send the response message to the client via the appropriate socket. Once the transaction is complete, the proxy should close the connection.

### **2.2 HTTP Proxies**

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software). However, in some circumstances it may be useful to introduce an intermediate entity called a proxy. Conceptually, the proxy sits between the client and the server. In the simplest case, instead of sending requests directly to the server the client sends all its requests to the proxy. The proxy then opens a connection to the server, and passes on the client's request. The proxy receives the reply from the server, and then sends that reply to the VM to check if that website is malicious or not. Once this decision is relayed back to the proxy, we send the stored data (the HTML file) back to the client. Notice that the proxy is essentially acting like both a HTTP client (to the remote server) and a HTTP server (to the initial client).

Why use a proxy? There are a few possible reasons:

* **Performance:** By saving a copy of the pages that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving a page, particularly if a server is remote or under heavy load.
* **Content Filtering and Transformation:** While in the simplest case the proxy merely fetches a resource without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving files. The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instances, by stripping out images to make a page easier to display on a handheld or other limited-resource client), or perform other transformations and filtering.
* **Privacy:** Normally, web servers log all incoming requests for resources. This information typically includes at least the IP address of the client, the browser or other client program that they are using (called the User-Agent), the date and time, and the requested file. If a client does not wish to have this personally identifiable information recorded, routing HTTP requests through a proxy is one solution. All requests coming from clients using the same proxy appear to come from the IP address and User-Agent of the proxy itself, rather than the individual clients. If a number of clients use the same proxy (say, an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

#### **Virtual Machine “Honeypot Browser”**

The virtual machine will be running a daemon process in the background continuously while listening on a specified port. The daemon will be waiting for a connection from the proxy. Once, it starts receiving buffer content from the proxy, it is stored in a HTML file in the local disk. The child process will spawn the browser with the help of the execvp () command which loads the particular HTML file. The parent process is running simultaneously and kills the child which shuts down the browser.

#### Extensions Implemented since the interim milestone:

#### **2.4 Caching**

Caching is one of the most common performance enhancements that web proxies implement. Caching takes advantage of the fact that most pages on the web don't change that often, and that any page that you visit once you (or someone else using the same proxy) are likely to visit again. A caching proxy server saves a copy of the files that it retrieves from remote servers. When another request comes in for the same resource, it returns the saved (or *cached*) copy instead of creating a new connection to a remote server. This saves a modest amount of time and CPU if the remote server is nearby and lightly trafficked, but can create more significant savings in the case of a more distant server or a remote server that is overloaded (it can also help reduce the load on heavily trafficked servers).

Caching introduces a few new complexities as well. First of all, a great deal of web content is dynamically generated, and as such shouldn't really be cached. Second, we need to decide how long to keep pages around in our cache. If the timeout is set too short, we negate most of the advantages of having a caching proxy. If the timeout is set too long, the client may end up looking at pages that are outdated or irrelevant.

Steps implemented to achieve the desired caching behavior for the web proxy:

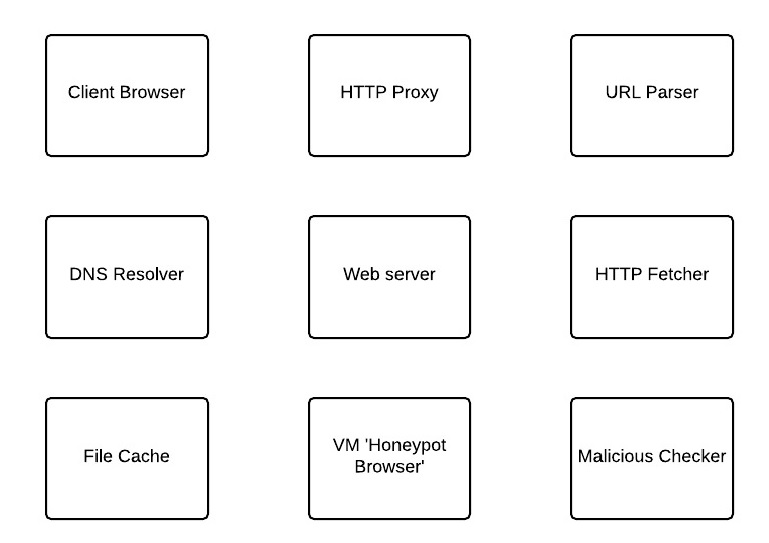
1. Read the “Date:” field using the strstr() function.
2. Check the “Host”, “If-Modified-Since” and “Connection: Close” fields and store the retrieved content in a buffer.
3. Once you start receiving content from the remote host, sscanf() was used read HTTP version and response code. If the response code is not 304, then create a cache entry by opening a file and naming it with the url-encoded.
4. Finally, the next time the same URL was sent as a request to the proxy, the url-encoded is checked in the cache first and uploaded if it exists, else start a new session with the remote server.

**2.5 Malicious Checking:**

The actual exploit checking will occur in the virtual machine. A ‘honeypot’ browser is forked from the child process in the running daemon. The browser replays the content it received from the proxy which is the website entered by the client in the client browser. Meanwhile, the parent process is running and checks the memory usage of the running browser using the system() command. Based, on the %memory used by the browser and analyzing it, a decision is made whether the website is malicious or not. This decision is returned to the proxy via the same socket connection it received the buffer data from.

**2.6 Modules:**

The following diagram explains the code flow with different modules:



1. **Testing**

**3.1 Proxying a client request:**

Initially, a socket connection is setup from the client browser to the proxy. The connection is accepted from the proxy and the HTTP request is processed. This is made sure it is valid by parsing it using the sscanf() function and checking it till the ‘/’ character. The type of HTTP request, the URL and the proto is recorded. The proxy makes sure if it is valid request by making sure we receive only GET requests. Here, strncmp is used to compare the string for GET and HTTP/1.1 or HTTP/1.0 requests. This URL is parsed to get the hostname which is located between http:// till “/”. This is done using he strtok() function. Using this hostname, we get the IP address by the gethostbyname() function. This is sent to the web server using a separate socket connection and record the HTTP response in a buffer and check the “Date:” field to store in the cache. Also, the connection is set to close using the sprint() function as “Connection: Close”. The buffer data that is received is checked till </html> tag and simultaneously sent to the VM and updating its cache contents as well.

**3.2 Virtual machine setup:**

We will set up a single virtual machine through virtualization software like VMware workstation. Let us call these machines VM1. Let the machine running the HTTP proxy be P. The client browser will be connected to P. P will be connected to the web server and also the VM1. All these connections are achieved using socket programming. This setup handles only HTTP requests. The VM1’s IP address and port are hard coded in P. There is a direct connection to VM1 to send the HTTP response data. The same connection is used to send the decision about maliciousness of the website. With this set up we can run the following test case to check the correctness of the web exploit scanner:

1) Send the buffer data received from the web server to the VM daemon.

2) Store the buffer data as a HTML file using basic file operations like fopen, fseek, fwrite.

3) Spawn the browser using the execvp command in the child process.

4) Check the %memory usage of the running browser by using the system command.

5) In the system command, checking is done by top –n 1, grep and sscanf for particular Memory values like shared memory, virtual memory or the RES (non swapped physical) memory.

6) These memory values are checked inside the parent process where a sleep command of 10 seconds runs between them to allow the child a process to spawn the browser for that much amount of time. If the difference between the memory values is greater than 30, then we relay back malicious decision else non-malicious.

**3.3 Relaying the response back from the VM**

The parent process which checks for maliciousness in the VM relays the decision back to the proxy using the same socket as with which the buffer was received. Depending on the response received from the VM, the proxy either uploads the “malicious.html” file or the non-malicious/ original file which is stored in the cache to the client browser. This is done with both the files being converted to buffer data using fseek(), malloc and fread and free operations and then sent to the client browser.

**3.4 Discriminate malicious from non-malicious domains:**

Pseudo code to check for maliciousness:

* *While the daemon process in the Virtual machine is listening on a particular socket*
* *Receive the <html> buffer data from the proxy and store it as a html page using the fwrite command.*
* *Fork a child using the fork() command*
* *Spawn the browser with that html page using the exevp() command in the child process*
* *Run the parent process and check for memory consumption of the RES(non-swapped physical memory) using the top command. Extract this value using the sscanf() function and store as a integer using the atoi function.*
* *Sleep for 10 seconds to allow the browser to continue to run*
* *Check the RES memory in the top command again now and calculate the difference from the previous value.*
* *If the difference is greater than 30(I observed this pattern only for malicious domains), then return the decision that it is malicious as most of the websites which have bad corrupted code contain JavaScript or the iFrame tags and because HTML markup can introduce malicious contents, the browser could interpret some data values as HTML tags or script and render it in the browser and run it continuously and consume more memory. usually will start consuming more residual physical memory within certain time difference(in my case it is 10 seconds as I wanted to check for memory explosions in the JavaScript obfuscated code)*
* *If the difference is less than 30, return the decision that it is not malicious.*

Observations made:

* The heap spray code(used in assignment) used consumed around 29% memory which sent a decision(m=13) that it was malicious, whereas a normal [www.wikipedia.org](http://www.wikipedia.org/) consumed only 8.8% memory usage and decision was sent back to the HTTP proxy. When the browser is running normally without any NOP sled, it consumes around 91608KB. When the heapspray code was run, it consumed 295608KB. Hence, a malicious alert was sent.
* The browser was successfully killed using SIGTERM command in the parent process and a memory check was again performed.

These are just some of the test cases running. We can similarly run all these tests for more malicious domains as well. We can also write some more tests to check the more corner case that a website can genuinely use more memory and raise a false exception which can be encountered in a real life system.

1. **Conclusion and Future Work:**

This document describes the architecture of a web exploit scanner. There are many improvements that can be done in the current architecture to make this exploit scanner work better. The following improvements can be implemented on this project:

* HTTPS 1.1 Support handling
* HTTP Connection Keep-Alive handling
* Make the exploit scanner completely *multithreaded*. We can use thread pooling and assign every GET request to a separate thread of its own. This would need a synchronized thread pool. This would make the request and response processing faster and concurrent. If time permits I would try to modify the current architecture to incorporate this idea.

#### Content Transformation:

Content transformation is the process of a proxy inserting, removing, or changing the contents of a resource requested from a remote server. Since the data returned from a web server is usually just text, this means that we can change the page almost any way we want- add or remove dirty words, change the text to Pig-Latin, rotate the images on the page 90 degrees, etc.

#### Link Prefetch:

If a user asks for a particular page, the odds are that he or she will next request a page linked from that page. Link prefetching uses this information to attempt to speed up browsing by parsing requested pages for links, and then fetching the linked pages in the background. The pages fetched from the links are stored in the cache, ready to be served to the client when they are requested without the client having to wait around for the remote server to be contacted.

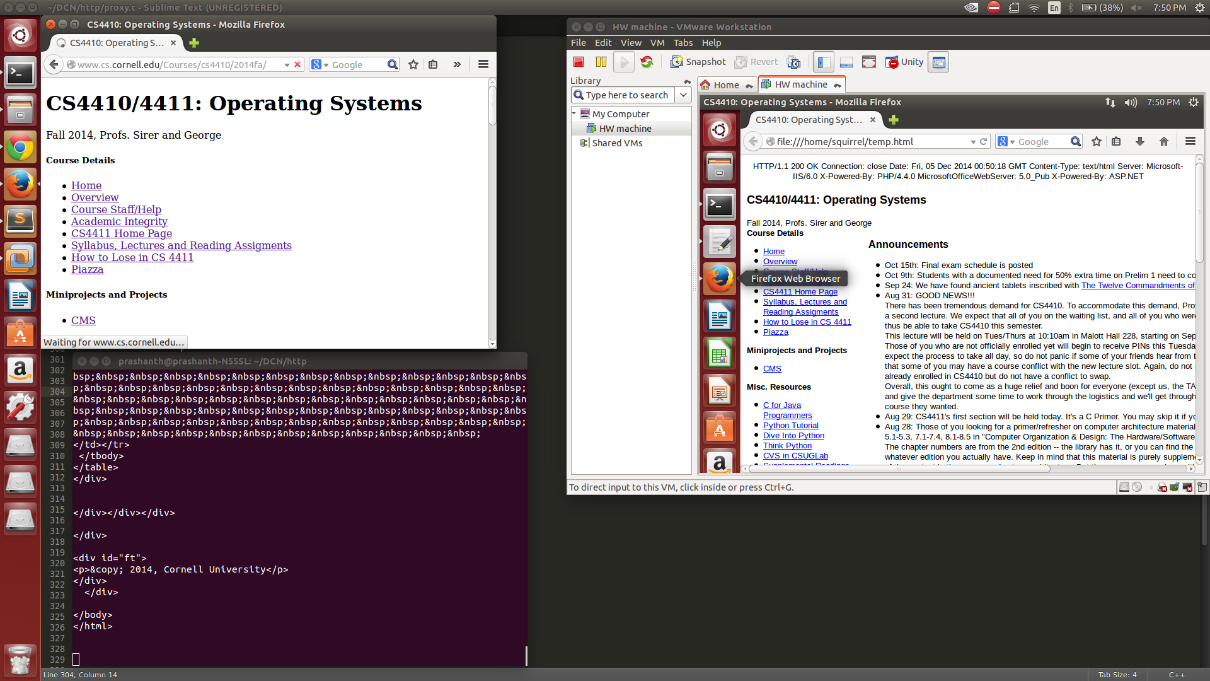
Parsing and fetching links can take an appreciable amount of time, especially for a page with a lot of links. Therefore we should make our proxy into a multi-threaded application. One thread should remain dedicated to the tasks that you have already implemented: reading requests from the client and serving pages from either the cache or a remote server. In a separate thread, the proxy will parse a page and extract the HTTP links, request those links from the remote server, and add them to the cache.

References:

* <http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html>
* <http://www.uwo.ca/its/doc/courses/notes/socket/>
* <http://www.jmarshall.com/easy/http/>
* StackOverflow for converting files to buffer data and vice versa

**Screenshots:**

This shows the content being sent to the VM and content being displayed in the VM and also the same content being displayed at the client browser:



This shows the screenshot of displaying a preloaded malicious html file:

