Monitoring HTTP Traffic with Bro

Bro can be used to log the entire HTTP traffic from your network to the http.log file. This file can then be used for analysis and auditing purposes.

In the sections below we briefly explain the structure of the http.log file, then we show you how to perform basic HTTP traffic monitoring and analysis tasks with Bro. Some of these ideas and techniques can later be applied to monitor different protocols in a similar way.

Introduction to the HTTP log

The http.log file contains a summary of all HTTP requests and responses sent over a Bro-monitored network. Here are the first few columns of `http.log` :

```
# ts           uid           orig_h         orig_p  resp_h          resp_p
1311627961.8   HSH4uV8KVJg   192.168.1.100  52303   192.150.187.43  80
```

Every single line in this log starts with a timestamp, a unique connection identifier (UID), and a connection 4-tuple (originator host/port and responder host/port). The UID can be used to identify all logged activity (possibly across multiple log files) associated with a given connection 4-tuple over its lifetime.

The remaining columns detail the activity that's occurring. For example, the columns on the line below (shortened for brevity) show a request to the root of Bro website:

```
# method   host         uri   referrer   user_agent
GET        bro.org      /     -          <...>Chrome/12.0.742.122<...>
```

Network administrators and security engineers, for instance, can use the information in this log to understand the HTTP activity on the network and troubleshoot network problems or search for anomalous activities. We must stress that there is no single right way to perform an analysis. It will depend on the expertise of the person performing the analysis and the specific details of the task.

For more information about how to handle the HTTP protocol in Bro, including a complete list of the fields available in http.log, go to Bro's HTTP script reference.

Detecting a Proxy Server

A proxy server is a device on your network configured to request a service on behalf of a third system; one of the most common examples is a Web proxy server. A client without Internet access connects to the proxy and requests a web page, the proxy sends the request to the web server, which receives the response, and passes it to the original client.

Proxies were conceived to help manage a network and provide better encapsulation. Proxies by themselves are not a security threat, but a misconfigured or unauthorized proxy can allow others, either inside or outside the network, to access any web site and even conduct malicious activities anonymously using the network's resources.

TABLE OF CONTENTS

NEXT PAGE

Bro IDS

PREVIOUS PAGE

Bro Logging

SEARCH

Search

What Proxy Server traffic looks like

In general, when a client starts talking with a proxy server, the traffic consists of two parts: (i) a GET request, and (ii) an HTTP/ reply:

```
Request: GET http://www.bro.org/ HTTP/1.1
Reply:   HTTP/1.0 200 OK
```

This will differ from traffic between a client and a normal Web server because GET requests should not include “http” on the string. So we can use this to identify a proxy server.

We can write a basic script in Bro to handle the http_reply event and detect a reply for a `GET http://` request.

http_proxy_01.bro

```
event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( /^[hH][tT][tT][pP]:/ in c$http$uri && c$http$status_code == 200 )
                print fmt("A local server is acting as an open proxy: %s", c$id$resp_h);
        }
```

```
# bro -r http/proxy.pcap http_proxy_01.bro
A local server is acting as an open proxy: 192.168.56.101
```

Basically, the script is checking for a “200 OK” status code on a reply for a request that includes “http:” (case insensitive). In reality, the HTTP protocol defines several success status codes other than 200, so we will extend our basic script to also consider the additional codes.

http_proxy_02.bro

```
module HTTP;

export {

        global success_status_codes: set[count] = {
                200,
                201,
                202,
                203,
                204,
                205,
                206,
                207,
                208,
                226,
                304
        };
}

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( /^[hH][tT][tT][pP]:/ in c$http$uri &&
             c$http$status_code in HTTP::success_status_codes )
```

```
                print fmt("A local server is acting as an open proxy: %s", c$id$resp_h);
        }
```

```
# bro -r http/proxy.pcap http_proxy_02.bro
A local server is acting as an open proxy: 192.168.56.101
```

Next, we will make sure that the responding proxy is part of our local network.

http_proxy_03.bro

```
@load base/utils/site

redef Site::local_nets += { 192.168.0.0/16 };

module HTTP;

export {

        global success_status_codes: set[count] = {
                200,
                201,
                202,
                203,
                204,
                205,
                206,
                207,
                208,
                226,
                304
        };
}

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( Site::is_local_addr(c$id$resp_h) &&
             /^[hH][tT][tT][pP]:/ in c$http$uri &&
             c$http$status_code in HTTP::success_status_codes )
                print fmt("A local server is acting as an open proxy: %s", c$id$resp_h);
        }
```

```
# bro -r http/proxy.pcap http_proxy_03.bro
A local server is acting as an open proxy: 192.168.56.101
```

> **Note**
>
> The redefinition of `Site::local_nets` is only done inside this script to make it a self-contained example. It's typically redefined somewhere else.

Finally, our goal should be to generate an alert when a proxy has been detected instead of printing a message on the console output. For that, we will tag the traffic accordingly and define a new `Open_Proxy` `Notice` type to alert of all tagged communications. Once a notification has been fired, we will further suppress it for one day. Below is the complete script.

http_proxy_04.bro

```
@load base/utils/site
@load base/frameworks/notice

redef Site::local_nets += { 192.168.0.0/16 };

module HTTP;

export {

        redef enum Notice::Type += {
                Open_Proxy
        };

        global success_status_codes: set[count] = {
                200,
                201,
                202,
                203,
                204,
                205,
                206,
                207,
                208,
                226,
                304
        };
}

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( Site::is_local_addr(c$id$resp_h) &&
             /^[hH][tT][tT][pP]:/ in c$http$uri &&
             c$http$status_code in HTTP::success_status_codes )
                NOTICE([$note=HTTP::Open_Proxy,
                        $msg=fmt("A local server is acting as an open proxy: %s",
                                 c$id$resp_h),
                        $conn=c,
                        $identifier=cat(c$id$resp_h),
                        $suppress_for=1day]);
        }
```

```
# bro -r http/proxy.pcap http_proxy_04.bro
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     notice
#open     2018-05-23-00-22-33
#fields   ts      uid     id.orig_h       id.orig_p       id.resp_h
#types    time    string  addr    port    addr    port    string  strin
1389654450.449603 CHhAvVGS1DHFjwGM9       192.168.56.1    52679   192.1
#close    2018-05-23-00-22-33
```

Note that this script only logs the presence of the proxy to `notice.log`, but if an additional email is desired (and email functionality is enabled), then that's done simply by redefining `Notice::emailed_types` to add the `Open_proxy` notice type to it.

Inspecting Files

Files are often transmitted on regular HTTP conversations between a client and a server. Most of the time these files are harmless, just images and some other multimedia content, but there are also types of files, specially executable files, that can damage your system. We can instruct Bro to create a copy of all files of certain types that it sees using the File Analysis Framework (introduced with Bro 2.2):

file_extraction.bro

```
global mime_to_ext: table[string] of string = {
        ["application/x-dosexec"] = "exe",
        ["text/plain"] = "txt",
        ["image/jpeg"] = "jpg",
        ["image/png"] = "png",
        ["text/html"] = "html",
};

event file_sniff(f: fa_file, meta: fa_metadata)
        {
        if ( f$source != "HTTP" )
                return;

        if ( ! meta?$mime_type )
                return;

        if ( meta$mime_type !in mime_to_ext )
                return;

        local fname = fmt("%s-%s.%s", f$source, f$id, mime_to_ext[meta$mime_type]);
        print fmt("Extracting file %s", fname);
        Files::add_analyzer(f, Files::ANALYZER_EXTRACT, [$extract_filename=fname]);
        }
```

```
# bro -r http/bro.org.pcap file_extraction.bro
Extracting file HTTP-FiIpIB2hRQSDBOSJRg.html
Extracting file HTTP-FMG4bMmVV64eOsCb.txt
Extracting file HTTP-FnaT2a3UDd093opCB9.txt
Extracting file HTTP-FfQGqj4Fhh3pH7nVQj.txt
Extracting file HTTP-FsvATF146kf1Emc21j.txt
[...]
```

Here, the `mime_to_ext` table serves two purposes. It defines which mime types to extract and also the file suffix of the extracted files. Extracted files are written to a new `extract_files` subdirectory. Also note that the first conditional in the `file_new` event handler can be removed to make this behavior generic to other protocols besides HTTP.

Internal Pages