Web Application Attack Analysis Using Bro IDS

Ganesh Kumar
WEBSITE APPLICATION ATTACK ANALYSIS USING BRO IDS

GIAC (GCIA) Gold Certification

Author: Ganesh Kumar Varadarajan, ganeshkumar.varadarajan@gmail.com
Advisor: Manuel Humberto Santander Peláez, manuel@santander.name

Accepted: 15 Oct 2012

Abstract

"The purpose of the paper is to analyze the effectiveness of Bro IDS in detecting web application attacks. In order to detect known web-based attacks, intrusion detection systems are usually equipped with a large number of signatures. They can however be fooled by obfuscated input techniques and allow the query to pass unfiltered to the web application. The paper will explore the use of application layer knowledge of data as well as signatures to detect common web attacks using Bro IDS scripting language."
1. Introduction

Bro is an open-source, Unix-based Network Intrusion Detection System (NIDS) that passively monitors network traffic and looks for suspicious activity. Bro detects intrusions by first parsing network traffic to extract its application-level semantics and then executing event-oriented analyzers that compare the activity with patterns deemed troublesome (Richard, 2005). Its analysis includes detection of specific attacks including those defined by signatures, but also those defined in terms of events and unusual activities (e.g., certain hosts connecting to certain services, or patterns of failed connection attempts).

Bro uses a specialized policy language that allows a site to tailor Bro's operation, both as site policies evolve and as new attacks are discovered. If Bro detects something of interest, it can be instructed to either generate a log entry, alert the operator in real-time, execute an operating system command (e.g., to terminate a connection or block a malicious host on-the-fly) (Babbin, 2006). In addition, Bro's detailed log files can be particularly useful for forensics.

2. Web Attack Intrusion Detection

The important feature of bro that differentiates it from other IDS systems such as SNORT is that bro scripts could be written to understand application semantics and could be trained to look for anomalies which can effectively eliminate attacks as compared to pattern oriented rules found in systems such as SNORT (Jacob, 2006). SNORT is a signature based intrusion detection system which relies on the availability of good signatures (patterns) to detect intrusions. A pattern could be similar to a HTTP request containing c:\boot.ini to a windows web server or /etc/passwd for a linux web server. In a signature based detection system, the observed packets are matched using available signatures using regular expressions. Thus the quality of detection is based on the quality of the signature base whereas Bro is an anomaly based intrusion detection system that matches the observed packets with the desired application profile. For example, an alert could be triggered if multiple attempts are made by the user within a short time against
the application. This is an application profile. A bro script could be written to keep track of user attempts against the application and trigger an alert if it exceeds a threshold value. This requires the intrusion detection system to not only understand the protocol but also keep track of failed user sessions against the application. This crucial feature of Bro to understand the higher order application details gives it a distinct advantage against signature based intrusion detection systems.

Most often attacks can sneak through Signature based detection systems. For example, if XSS attacks are considered, IDS systems most often look for presence of start of script characters. This could be easily fooled by using different encoding methods such as encoding special characters using variety of encoding methods (URL, base64 etc) and which would defeat the IDS filters and attack the application. A polymorphic XSS worm is such an example and can defeat a signature based intrusion detection system. If Bro is used as an intrusion detector, a script could potentially be written which would look for non native characters to the application form field and send an alert notice indicating a potential intrusion activity. Thus because of higher level knowledge of application profile, complex intrusion activity such as polymorphic worms can be detected quickly compared to traditional systems.

2.1. BRO Scripting

This section will give a basic introduction in to writing bro scripts using bro scripting language. It is not intended as a complete reference and will serve to explain the bro scripts used in attack detection in the later sections.

Bro can detect a large number of protocols, and the notice policy tells which of them the user wants to be acted upon in some manner. In particular, the notice policy can customize the specific actions that needs to be taken, such as sending an alert to the Security Incident and Event Management (SIEM) framework or adding firewall rules to block the offending IP’s. Bro ships with a large number of policy scripts which perform a wide variety of analyses (Bro Documentation, 2012). Both network and application attacks can be detected using Bro scripts though there is some customizing that is needed
to suit your environment. Bro gives a lot of tools that will simplify the task. But to detect a actual attack, a local script needs to be written for your environment.

The policy scripts are prewritten scripts that are included for variety of protocols such as HTTP, SSH, FTP, DNS, SMTP etc and a variety of scripts for filtering and post processing such as for logging, reporting and alerting. By default, these will be installed into $PREFIX/share/bro and can be identified by the use of a .bro file name extension (Bro Documentation, 2012). The main entry point for a standalone Bro instance managed by BroControl is the $PREFIX/share/bro/site/local.bro script. This script can be modified to suite the environment.

The local configuration file ( local.bro) needs to specify which activity is actionable based on the results of the analysis flagged by the policy scripts. A very simple bro script is as follows

```plaintext
Global attack_count = 0;
event connection_established(c: connection) {
    if (c$id$orig_h == 1.1.1.1 &&
c$id$resp_p == 313/tcp &&
++attack_count == 5 )
        NOTICE([$note=Attack,
        $conn=c,
        $msg=fmt("Attack from %s to destination: %s", c$id$orig_h, c$id$resp_h)];
}
```

The above script basically generates a notice (A custom log message generated by Bro to indicate events of interest) whenever a host 1.1.1.1 makes 5 successful connections to port 313/tcp.

One of the common entries used in a bro script is the “redef enum Notice::Type += {“. The “+= “operator allows to add onto an already defined variable. In the case a value is added to the enumerable constant Notice::Type (Ryesecurity, 2012). Different Notice types such as “XSS Injection Attack” or “SQL Injection Attack” thus can be added to customize the Bro Notice for easier readability.

Attributes occur at the end of type/event declarations and change their behavior. The syntax for declaring attributes is &var or &var=val. Some of the major attributes in Bro language are

Author Name, email@address
&redef: Allows for redefinition of initial object values. This is typically used with constants, for example, const clever = T &redef; would allow the constant to be redefined at some later point during script execution (Bro Documentation, 2012).

&default: Uses a default value for a record field or container elements. For example, table[int] of string &default="foo" } would create a table that returns the string "foo" for any non-existing index (Bro Documentation, 2012).

&persistent: Makes a variable persistent, i.e., its value is written to disk (per default at shutdown time) (Bro Documentation, 2012).

The Bro scripting language supports different built-in types such as void, bool, int, count, counter, double, time, interval, string, pattern, enum, timer, port, addr, subnet, any, table, set, vector, record, file, func and event (Bro Documentation, 2012). Function types in Bro are declared using “function (argument*): type”. The argument is a (possibly empty) comma-separated list of arguments, and type is an optional return type. Event handlers are nearly identical in both syntax and semantics to a function, with the differences being that event handlers have no return type since they never return a value, and you cannot call an event handler. An event handler is usually executed either from a event engine or from a event statement in the script or from the schedule statement in the script.

A simple HTTP analysis script in Bro language is shown below

```bro
module HTTP;

export {
  redef enum Notice::Type += {
    ## Generated if a Command injection takes place using URL
    URI_Injection
  }
}

event http_header(c: connection, is_orig: bool, name: string, value: string)
{
  if (/AUTHORIZATION/ in name && /Basic/ in value)
  {
    local parts: string_array;
    parts = split1(decode_base64(sub_bytes(value, 7, |value|)), /:/);
    NOTICE($note=HTTP::Basic_Auth_Server,$msg=fmt("username: %s password: %s", parts[1], HTTP::default_capture_password == F ? "Blocked" : parts[2]),$conn=c);
  }
}
```

Author Name, email@address
In the script above, the appearance of HTTP basic Authentication in the HTTP request header is detected and flagged to the alert log. The key point is the availability of events such as event `http_header(c: connection, is_orig: bool, name: string, value: string)`, event `http_request(c: connection, method: string, original_URI: string, unescaped_URI: string, version: string)`, event `http_entity_data(c: connection, is_orig: bool, length: count, data: string)` which can be used for creating very targeted notices of interest to the user. Within these event handlers, customized pattern matching can take place to detect events or variables can be used to rank the pattern against a database to give a score to the pattern. By using such techniques an advanced detection script can be developed which can be used in detecting attacks. Bro can also be used for detecting other forms of authentication such as digest authentication and form authentication. In digest authentication, the client sends the GET request as follows

```
GET /dir/index.html HTTP/1.0
Host: localhost
Authorization: Digest username="admin",
    realm="admin@test.com",
    nonce="deefgeghf36594373131",
    uri="/dir/test.html",
    qop=auth,
    nc=00000001,
    cnonce="0e4f323c",
    response="48845fae49393f05355450972504c4abc",
    opaque="48593ehff23336773t"
```

To detect this type of authentication, the event handler script could be written as follows

```
event http_header(c: connection, is_orig: bool, name: string, value: string)
{
    if (/AUTHORIZATION/ in name && /Digest/ in value)
    {
        // filter response values and Server response
    }
}
```

Bro relies primarily on its scripting language for detecting events of interest. However there is also a pattern matching template called signatures which is similar to Snort-style pattern matching.

Author Name, email@address
A typical signature looks as follows

```plaintext
Signature testsig {
  ip-proto == tcp
  dst-port == 80
  http-request /.*(boot.ini)/
  event "Found windows boot!"
}
```

Each individual signature has the format signature <id> { <attributes> }. <id> is a unique label for the signature. There are two types of attributes: conditions and actions. The conditions define when the signature matches, while the actions declare what to do in the case of a match.

In the above signature, the protocol and destination port are the header conditions, http-request is a content condition and the action (event) defines what to do if the signature matches. The content conditions perform pattern matching on elements extracted from an application protocol dialogue. For example, http-request /.*boot.ini/ scans http request headers requested within HTTP sessions. Note that for TCP connections, header conditions are only evaluated for the first packet from each endpoint. If a header condition does not match the initial packets, the signature will not trigger.

### 2.2. Test Setup

All the packet captures that were used in this paper were obtained through attacking a Virtual machine running Damn Vulnerable Web application and Web Goat (Refer Fig 1a). The request and responses from the virtual machine was captured using sniffer tool such as wireshark and analysis was performed using Bro IDS. The Virtual router in the diagram is a Linux host which is running virtual Box virtualization software. WebGoat and DVWA are run as Virtual box guests. Wireshark is made to run on the Linux host and this serves as a network tap station. An attacker system is present on a separate station and the packets sent to the VirtualBox guest has to pass the network interface on the Linux host. Thus packets destined for the Attacked system can be captured by wireshark.
To perform an attack, the attacking system uses a web attack tool BURP Intruder and Tamper Data (Firefox addon) which can be used to send customized HTTP request to the application. A sniffer tool is run on a separate system in the same VLAN as the Webgoat/DVWA application to capture all the request/responses. To perform an XSS attack, the attacker modifies the GET / POST request to exploit server side code to steal information from the client browsers. In SQL injection attack, the attacker modifies the GET/POST requests so that SQL scripts could be run on the server side system so that sensitive information could be dumped from that system. In both cases, the method of attack involves modifying the HTTP GET/POST parameters and this can be fully captured in the network sniffer that is running on a separate station.

The attack dumps for Webgoat and DVWA application are shown below

![Figure 1: XSS attack in DVWA (Get request)](image-url)

Figure 1: XSS attack in DVWA (Get request)
Figure 2: XSS Attack in WebGoat (POST request)

The Figures 1 and 2 shows the attack against IP 192.168.245.1 from the attacking system 192.168.2.x. It can be seen that name parameter is fuzzed with javascript input which will cause client side code to execute on 192.168.2.x.

Figure 3: SQL Injection Attack in WebGoat (POST request)

Figure 4: SQL Injection Attack in DVWA (Get request)

Figure 5: SQL Injection Attack in DVWA (Get request)

Author Name, email@address
2.3. Signature detection

In signature-based detection, alarms are generated based on specific attack signatures. These attack signatures encompass specific traffic or activity that is based on known intrusive activity.

2.3.1. Reflected XSS Injection

In the DVWA application, the name parameter is susceptible to XSS injection requests. In figure 1, packets 406,448 and 523 malicious input being sent to the DVWA application. Packet 38 even though having character “<” is not malicious and is a normal input to the application. In Web Goat application, the POST parameter “search_name” is susceptible to XSS attack as shown in figure 2.

A typical Bro signature to detect XSS attack is as follows:

```plaintext
signature xss-sig {
ip-proto == tcp
dst-port == 80
http-request /.*(<)/
event "Found XSS!"
}
signature xss-sig2 {
ip-proto == tcp
dst-port == 80
http-request-body /.*([a-zA-Z0-9-_])/ 
  event "Found XSS in BODY!"
}
```

Figure 8: XSS Bro Signature
The signature for the http-request is pretty broad with a pattern match looking for any start of script character while for the http request body; it is more directed to catch URL encoded characters. The http-request body is purposely kept more directed as it is pretty easy to flag a variety of input as XSS vectors whereas actually it would be harmless input.

If the above signature is tested with the packet capture shown in figure 1 and 2, the result obtained is as shown in figure 9 and 10. The result in figure 1 shows that all the requests have been identified as potential XSS vectors. The first request is clearly a name which has been entered incorrectly with an additional symbol and which has been flagged as XSS vector by the Bro IDS signature. For the Web goat application, the correct vector has been identified. The problem with the signature is that, it is looking for a particular signature namely URL encoded characters. Not all applications will have the same characteristics and it would be pretty simple to defeat this signature by encoding in other formats such as base64 or plain ascii text.

2.3.2. SQL Injection

SQL injection is a code injection technique that exploits security vulnerabilities in website's software. The vulnerability happens when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is
not strongly typed and unexpectedly executed. SQL commands are thus injected from the web form into the database of an application (like queries) to change the database content or dump the database information like credit card or passwords to the attacker.

The characteristic feature that is found in SQL injection requests (See figures 3, 4 and 5) is the presence of SQL escape character "'". This can be used in SQL detection signature in Bro as follows

```plaintext
signature sql-sig {
  ip-proto == tcp
  dst-port == 80
  http-request /.*(['\'])/ event "Found SQLinjection!"
}

signature sql-sig3 {
  ip-proto == tcp
  dst-port == 80
  http-request-body /.*([^a-zA-Z0-9=&<>_])/ event "Found sqlinjection in BODY!"
}
```

The signature basically looks for presence of the literal escape character usually used to injection to add additional SQL statements to web forms. The effectiveness of the signatures is shown in figures below.

Figure 11: SQL injection Bro Signature

Figure 12: SQL Signature alert for Web Goat application
But the same signature would not be able to detect Blind sql injection example shown in figure 7 because of the absence of SQL escape character. As shown in figure 7, the blind sql injection statement used is “5+and+substring(@@version,1,1)=5”. Thus to detect this injection, further drilling down is required. The signature could be further tuned to detect such type of attacks, but it may prove to be ineffective in a enterprise environment containing hundreds of applications requests containing different data inputs which may match the SQL injection statements or characters used in the signature.

2.4. Anomaly detection

With anomaly detection, a profile is created of each input on your system. These profiles can be built automatically or created manually. How the profiles are created is not important as long as the profiles accurately define the characteristics for each input of the web application being monitored. These profiles are then used as a baseline to define normal user activity. If any network activity deviates too far from this baseline, then the activity generates an alarm. Because this type of IDS is designed around profiles, it is also sometimes known as profile-based detection (Ryan, 2009).

2.4.1. Reflected XSS Injection

The attack can be detected by writing an application aware script shown in figure 6. Parameters of interest can be profiled for this request. If string length and presence of alphanumeric characters is taken as a measure of anomaly for this request, The
application can have two parameters of interest that would characterize if the input is valid. One if the string length and another is the presence of character “<” or “>” in the parameter of interest which is used to inject client side script.

```javascript
module HTTP;
export {
    redef enum Notice::Type += {
        XSS_URI_Injection_Attack,
        XSS_Post_Injection_Attack,
    };

    ## URL message input
    type UMessage: record
        {text: string;       ##< The actual URL body}
    ;

    const match_xss_uri = /[<>/]/ &redef;
    const match_xss_uri1 = /[<>/]/ &redef;
    const match_xss_body = /[<>]/ &redef;
    global ascore:count &redef;
    global http_body:string &redef;

    redef record Info += {
        ## Variable names extracted from all cookies.
        post_vars: vector of string &optional &log;
    };

    ### parse body
    function parse_body(data: string)  : UMessage
    {
        local msg: UMessage;
        local array = split(data, /search_name=/);
        for( i in array)
        {
            local val = array[i];
            msg$text = val;
        }

        if( i == 2)
        {
            return msg;
        } else
        {
            msg$text = "";
            return msg;
        }
    }
```
## Parse URI

```javascript
function parse_uri(data: string): UMessage {
  local msg: UMessage;
  local array = split(data, /name=/);
  for (i in array) {
    local val = array[i];
    msg$text = val;
  }
  if (i == 2) {
    return msg;  // returns msg
  } else {
    msg$text = "";
    return msg;
  }
}
```

### GET XSS IN REQUEST BODY

```javascript
msg = parse_body(http_body);
if (byte_len(msg$text) > 10)
  ++ascore;
if (match_xss_body in msg$text)
  ++ascore;
if (ascore >= 3)
  NOTICE([Snote=XSS_Post_Injection_Attack,
    Sconn=c,
    $msg=fmt("XSS Attack from %s to destination: %s with Attack string %s and post data %s",
    c$SidSorig_h, c$SidSresp_h, c$HttpSuri, http_body)])
```

### GET XSS IN HTTP REQUEST HEADER

```javascript
# GET XSS IN HTTP REQUEST HEADER
```

Author Name, email@address
msg = parse_uri(c$http$uri);

# Test for string length
if ( byte_len(msg$text) > 10)
    ++ascore;
if(match_xss_uri in msg$text)
    ++ascore;
if(match_xss_uri1 in msg$text)
    ++ascore;
if ( ascore >= 3)
    {
        NOTICE(($note=XSS_URI_Injection_Attack,
            $conn=c,
            $msg=fmt("XSS Attack from %s to destination: %s with Attack string %s", c$id$orig_h, c$id$resp_h, c$http$uri));
    }

Figure 15: Bro script for detecting XSS

When the XSS detection script is run against DVWA application, bro generate a notice log detailing the attack vector in the log.

With WebGoat, the following result was obtained,

Notice that in both cases, the attack vector has been correctly identified. The advantage of anomaly detection lies in the fact that the IDS rule can be tuned for the application by looking at parameters of interest and alerts the admin against such attacks.

Author Name, email@address


2.4.2. SQL Injection

The attack can be detected by writing an application aware script shown in figure 6. Parameters of interest can be profiled for this request. If string length and presence of SQL escape character is taken as a measure of anomaly for this request, the application can have two parameters of interest that would characterize if the input is valid. One if the string length and another is the presence of character “‘” in the parameter of interest which is used to inject client side SQL. We have seen that the characterizing based on presence of SQL escape character alone would lead to missing blind SQL injection scenarios as seen in figure. To detect those injections, a combination metric which can increase the anomaly score if the byte length is greater than a minimum value and presence of numeric characters could be used to detect more complex injections.

```plaintext
# Anomaly detection of SQL attacks (Ryesecurity, 2012)
@load base/frameworks/notice
@load base/protocols/ssh
@load base/protocols/http

module HTTP;
export {
    redef enum Notice::Type += {
        SQL_URI_Injection_Attack,
        SQL_Post_Injection_Attack,
    };

    ## URL message input
    type UMessage: record
    {
        text: string;       ##< The actual URL body
    };

    const match_sql_uri = /[']]/ &redef;
    const match_sql_uri1 = /[']]/ &redef;
    const match_sql_uri2 = /[0-9]/ &redef;
    const match_sql_body = /[']]/ &redef;

    global ascore:count &redef;
    global http_body:string &redef;

    redef record Info += {
        ## Variable names extracted from all cookies.
        post_vars: vector of string &optional &log;
    };
}

### parse body

function parse_body(data: string) : UMessage
{
    local msg: UMessage;

    local array = split(data, /password=/);
    ...
```

Author Name, email@address
for( i in array)
{
    local val = array[i];
    msg$text = val;
}

if( i == 2)
{
    return msg;  # returns msg
}
else
{
    msg$text = "";
    return msg;
}

## Parse URI
function parse_uri(data: string) : UMessage
{
    local msg: UMessage;
    local array = split(data, /id=/);
    for ( i in array)
{
    local val = array[i];
    msg$text = val;
}

if(i == 2)
{
    return msg;  # returns msg
}
else
{
    msg$text = "";
    return msg;
}

Event http_entity_data(c: connection, is_orig: bool, length: count, data: string) &priority=5
{
    local msg:UMessage;
    ascore = 1;
    if(c$http$first_chunk)
{
    http_body = data;

    ## GET SQL IN REQUEST BODY

    msg = parse_body(http_body);
    if(byte_len(msg$text) > 10)
        ++ascore;
    if(match_sql_body in msg$text)
    {
        ++ascore;
        if(match_sql_uri1 in msg$text)
event http_request(c: connection, method: string, original_URI: string,
            unescaped_URI: string, version: string) &priority=3
{
    local msg:UMessage;
    local body:UMessage;

    ascore = 1;

    # GET SQL IN HTTP REQUEST HEADER
    msg = parse_uri(c$http$uri);

    # Test for string length
    if ( byte_len(msg$text) > 2)
        ++ascore;

    if(match_sql_uri in msg$text)
    {
        ++ascore;

        if(match_sql_uri1 in msg$text)
            ++ascore;
    }

    if(match_sql_uri2 in msg$text && byte_len(msg$text) > 2)
    {
        ++ascore;
    }

    if ( ascore >= 3)
    {
        NOTICE({$note=SQL_URI_Injection_Attack,
                 $conn=c,
                 $msg=fmt("SQL Attack from %s to destination: %s with Attack string %s", c$ Sid$orig_h, c$ Sid$resp_h, c$ http$uri, c$http$Suri));
    }
}

Figure 18: Bro Script for detecting SQL injection
The result of applying the SQL detection script to the four SQL injection vectors are shown in figures 19, 20, 21 and 22. Notice that the SQL blind injection vector attempt has been detected because of the use of combination metrics used in detecting anamolies.

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>User</th>
<th>IP Address</th>
<th>Destination</th>
<th>Attack Type</th>
<th>Attack String</th>
<th>Post Data</th>
<th>Notice Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1339916569</td>
<td>YJfjLhYWLA9</td>
<td>192.168.245.1</td>
<td>192.168.245.128</td>
<td>HTTP::SQL_Post_Injection_Attack</td>
<td>/WebGoat/attack?Screen=213&amp;menu=1100</td>
<td>employee_id=112&amp;password=x'or'a'='a&amp;action=Login</td>
<td>Notice::ACTION_LOG 6 3600.000000 F</td>
</tr>
<tr>
<td>1339916791</td>
<td>q5z39ByqpB1</td>
<td>192.168.245.1</td>
<td>192.168.245.128</td>
<td>HTTP::SQL_URI_Injection_Attack</td>
<td>/dvwa/vulnerabilities/sqli/?id=1&amp;Submit=Submit</td>
<td>-</td>
<td>Notice::ACTION_LOG 6 3600.000000 F</td>
</tr>
<tr>
<td>1339916811</td>
<td>8CtiEfN7jG9</td>
<td>192.168.245.1</td>
<td>192.168.245.128</td>
<td>HTTP::SQL_URI_Injection_Attack</td>
<td>/dvwa/vulnerabilities/sqli/?id=1+or+'1'='1&amp;Submit=Submit</td>
<td>-</td>
<td>Notice::ACTION_LOG 6 3600.000000 F</td>
</tr>
<tr>
<td>1339919061</td>
<td>MgaH1dgw96c</td>
<td>192.168.245.1</td>
<td>192.168.245.128</td>
<td>HTTP::SQL_URI_Injection_Attack</td>
<td>/dvwa/vulnerabilities/sqli_blind/?id=1&amp;Submit=Submit</td>
<td>-</td>
<td>Notice::ACTION_LOG 6 3600.000000 F</td>
</tr>
<tr>
<td>1339919068</td>
<td>MgaH1dgw96c</td>
<td>192.168.245.1</td>
<td>192.168.245.128</td>
<td>HTTP::SQL_URI_Injection_Attack</td>
<td>/dvwa/vulnerabilities/sqli_blind/?id=1+and+substring(@@version,1,1)=4&amp;Submit=Submit</td>
<td>-</td>
<td>Notice::ACTION_LOG 6 3600.000000 F</td>
</tr>
</tbody>
</table>

Figure 19: Notice log for WebGoat application (SQL injection)

Figure 20: Notice log for DVWA application (SQL injection)

Figure 21: Notice log for DVWA application (SQL injection)

Figure 22: Notice log for DVWA application (SQL injection)
3. Conclusion

Through these tests using Bro IDS, what has been shown is that Bro IDS is able to perform application level deep packet inspection and it would be pretty easy to tune the application to generate alert logs for web vectors. It is a known fact that IDS signatures would generate false positives and this can be further fine tuned by generating notices by using Bro’s application inspection capability to a event monitoring systems such as splunk, sguil etc which can generate meaningful alerts regarding web attacks.

4. References


Figure 22: Notice log for DVWA application (SQL blind injection)