Interested in learning more about security?

SANS Institute InfoSec Reading Room

Web Application Attack Analysis Using Bro IDS

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.

WEB 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 a 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

```
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

&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 writen 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

```
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
            ]);
```

```
  }
}
```

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

```
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.

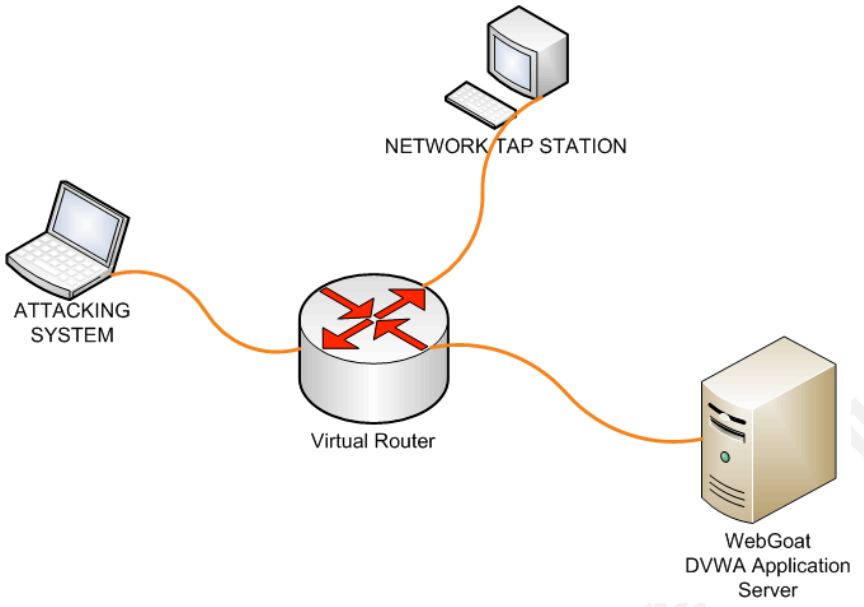

Fig 1a. Test setup.

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

No.	Time	Source	Destination	Protocol	Length	Info
38	2012-	192.168.2	192.168.245.128	HTTP	633	GET /dvwa/vulnerabilities/xss_r/?name=0%27Mallory%3C HTTP/1.1
41	2012-	192.168.2	192.168.245.1	HTTP	418	HTTP/1.1 200 OK (text/html)
406	2012-	192.168.2	192.168.245.128	HTTP	678	GET /dvwa/vulnerabilities/xss_r/?name=Hello+0%27Mallory%253Cscript%3Ealert%280%29%3C%2Fscript%3E HTTP/1.1
409	2012-	192.168.2	192.168.245.1	HTTP	434	HTTP/1.1 200 OK (text/html)
448	2012-	192.168.2	192.168.245.128	HTTP	720	GET /dvwa/vulnerabilities/xss_r/?name=Hello+0%27Mallory%3Cscript%3Ealert%280%29%3C%2Fscript%3E HTTP/1.1
451	2012-	192.168.2	192.168.245.1	HTTP	429	HTTP/1.1 200 OK (text/html)
523	2012-	192.168.2	192.168.245.128	HTTP	712	GET /dvwa/vulnerabilities/xss_r/?name=0%27Mallory%3Cscript%3Ealert%280%29%3C%2Fscript%3E HTTP/1.1
526	2012-	192.168.2	192.168.245.1	HTTP	428	HTTP/1.1 200 OK (text/html)
564	2012-	192.168.2	192.168.245.128	HTTP	640	GET /dvwa/security.php HTTP/1.1
571	2012-	192.168.2	192.168.245.1	HTTP	512	HTTP/1.1 200 OK (text/html)

Figure 1: XSS attack in DVWA (Get request)

```
Host: 192.168.245.128
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.245.128/WebGoat/attack?Screen=40&menu=900
Cookie: PHPSESSID=4t56i3mq18meb41civv9s3o3n4; acopendivids=phpbb2,redmine; acgroupswithpersist=nada; JSESSIONID=FC4B42A5BF6AD1D71FEC2D245
Authorization: Basic cm9vdDpvd2FzcGJ3YQ==
Content-Type: application/x-www-form-urlencoded
Content-Length: 70

search_name=%3Cscript%3Ealert%280%29%3C%2Fscript%3E&action=FindProfileHTTP/1.1 200 OK
Date: Fri, 15 Jun 2012 05:43:52 GMT
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
```

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

Stream Content

```
POST /WebGoat/attack?Screen=213&menu=1100 HTTP/1.1
Host: 192.168.245.128
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/20100101 Firefox/13.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.245.128/WebGoat/attack?Screen=213&menu=1100&stage=1
Cookie: JSESSIONID=8D3A461DB8397AE835D0C79C19A929BA; acopendivids=phpbb2,redmine;
acgroupswithpersist=nada
Authorization: Basic cm9vdDpvd2FzcGJ3YQ==
Content-Type: application/x-www-form-urlencoded
Content-Length: 48

employee_id=112&password=x'or'a'='a&action=LoginHTTP/1.1 200 OK
Date: Sun, 17 Jun 2012 07:02:49 GMT
```

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

Stream Content

```
GET /dvwa/vulnerabilities/sqli/?id=1%27+or+%271%27%3D%271&Submit=Submit HTTP/1.1
Host: 192.168.245.128
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/20100101 Firefox/13.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.245.128/dvwa/vulnerabilities/sqli/?id=efefe&Submit=Submit
Cookie: security=low; JSESSIONID=8D3A461DB8397AE835D0C79C19A929BA;
acopendivids=phpbb2,redmine; acgroupswithpersist=nada;
PHPSESSID=fg71m4bp9spbofqi29kqnqmpr4

HTTP/1.1 200 OK
Date: Sun, 17 Jun 2012 07:06:51 GMT
```

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

```
GET /dvwa/vulnerabilities/sqli/?id=%27+union+all+select+1%2C%40%40VERSION--+
+&Submit=Submit HTTP/1.1
Host: 192.168.149.128
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.149.128/dvwa/vulnerabilities/sqli/
Cookie: security=low; PHPSESSID=tk1ke8emvkjnkvdecb3va69u85; acopendivids=phpbb2,redmine;
acgroupswithpersist=nada

HTTP/1.1 200 OK
```

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

Author Name, email@address

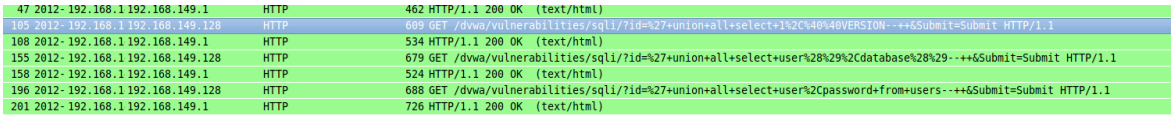

Figure 6: SQL Injection Attack in DVWA (Get request)

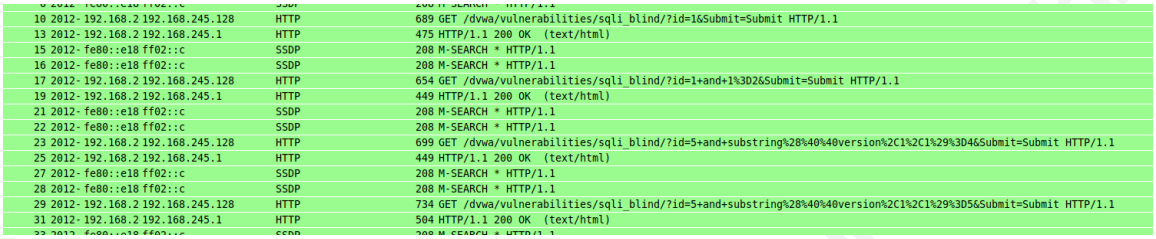

Figure 7: Blind SQL Injection Attack in DVWA (Get request)

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

```
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

Author Name, email@address

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.

```
1339731292.108612      192.168.245.1   49508   192.168.245.128 80      Signatures::Sensitive_Signature xss-sig
192.168.245.1: Found XSS!      /dvwa/vulnerabilities/xss_r/?name=O'Mallory<   -      -

1339731510.099221      192.168.245.1   49509   192.168.245.128 80      Signatures::Sensitive_Signature xss-sig
192.168.245.1: Found XSS!      /dvwa/vulnerabilities/xss_r/?name=Hello+0'Mallory%3Cscript>alert(0)</script>    -
-

1339731528.847472      192.168.245.1   49510   192.168.245.128 80      Signatures::Sensitive  Signature xss-sig
192.168.245.1: Found XSS!      /dvwa/vulnerabilities/xss_r/?name=Hello+0'Mallory<script>alert(0)</script>      -      -

1339731565.722829      192.168.245.1   49511   192.168.245.128 80      Signatures::Sensitive_Signature xss-sig
192.168.245.1: Found XSS!      /dvwa/vulnerabilities/xss  r/?name=0'Mallory<script>alert(0)</script>    -      -

1339731587.691317      192.168.245.1   49513   192.168.245.128 80      Signatures::Sensitive_Signature xss-sig
192.168.245.1: Found XSS!      /dvwa/vulnerabilities/xss_r/?name=0'Mallory<script>alert(0)</script>    -      -
```

Figure 9: Signature alert for DVWA application

```
Signatures::Sensitive_Signature xss-sig3        192.168.245.1: Found XSS in BODY!
search_name=%3Cscript%3Ealert%280%29%3C%2Fscript%3E&action=FindProfile  -      -
```

Figure 10: Signature alert for Webgoat application

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

```
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!"
}
```

Figure 11: SQL injection Bro Signature

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.

```
#types time   string addr   port   addr   port   enum   enum   string string addr   addr   port   count   string
table[enum]   table[count]   interval   bool   string string string double double addr   string subnet
1339916569.727439   9FzqCSOfuj8   192.168.245.1   51245   192.168.245.128 80   tcp
Signatures::Sensitive_Signature 192.168.245.1: Found sqlinjection in BODY!
employee_id=112&password=x'or'a'='a&action=Login   192.168.245.1   192.168.245.128 80   -   bro
Notice::ACTION_LOG   6   3600.000000   F   -   -   -   -
   -   -   -   -
1339916613.969753   xIIddz1ltB3   192.168.245.1   51254   192.168.245.128 80   tcp
Signatures::Sensitive_Signature 192.168.245.1: Found sqlinjection in BODY!
employee_id=112&password=neville%27&action=Login   192.168.245.1   192.168.245.128 80   -   bro
Notice::ACTION_LOG   6   3600.000000   F   -   -
```

Figure 12: SQL Signature alert for Web Goat application

```
#types time   string addr   port   addr   port   enum   enum   string string addr   addr   port   count   string
table[enum]   table[count]   interval   bool   string string string double double addr   string subnet
1339916791.542641   FONjgHMjw2b   192.168.245.1   51279   192.168.245.128 80   tcp
Signatures::Sensitive_Signature 192.168.245.1: Found SQLinjection!
/dvwa/vulnerabilities/sqli/?id=1'&Submit=Submit 192.168.245.1   192.168.245.128 80   -   bro
Notice::ACTION_LOG   6   3600.000000   F   -   -   -   -   -   -
   -   -
```

Author Name, email@address

```
1339916811.243118      Z0z50sTVSGb     192.168.245.1   51284   192.168.245.128 80      tcp
Signatures::Sensitive_Signature 192.168.245.1: Found SQLinjection!
/dvwa/vulnerabilities/sqli/?id='1'+or+'1'='1&Submit=Submit      192.168.245.1   192.168.245.128 80      -       bro
Notice::ACTION_LOG      6       3600.000000     F
```

Figure 13: SQL injection Alert for DVWA application

```
1335930732.614115      tBwwYdB8F1j     192.168.149.1   56142   192.168.149.128 80      tcp
Signatures::Sensitive_Signature 192.168.149.1: Found SQLinjection!
/dvwa/vulnerabilities/sqli/?id='+union+all+select+1,@@VERSION--++&Submit=Submit 192.168.149.1
192.168.149.128 80      -       bro     Notice::ACTION_LOG      6       3600.000000     F       -       -
        -       -       -       -       -       -
1335930753.759023      ysdIWo5nvii     192.168.149.1   56143   192.168.149.128 80      tcp
Signatures::Sensitive  Signature 192.168.149.1: Found SQLinjection!
/dvwa/vulnerabilities/sqli/?id='+union+all+select+user(),database()--++&Submit=Submit   192.168.149.1
192.168.149.128 80      -       bro     Notice::ACTION_LOG      6       3600.000000     F       -
        -       -       -       -       -       -       -
1335930768.479527      Jox3U2DyWOb     192.168.149.1   56145   192.168.149.128 80      tcp
Signatures::Sensitive_Signature 192.168.149.1: Found SQLinjection!
/dvwa/vulnerabilities/sqli/?id='+union+all+select+user,password+from+users--++&Submit=Submit    192.168.149.1
192.168.149.128 80      -       bro     Notice::ACTION_LOG      6       3600.000000     F
```

Figure 14: SQL injection Alert for DVWA 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.

```
## Anamoly detection of XSS attacks  ( RyeSecurity, 2012)

@load base/frameworks/notice
@load base/protocols/ssh
@load base/protocols/http

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 = /[3C3E]/ &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;
  }
```

Author Name, email@address

```
}

## Parse URI
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;
   }
 }



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 XSS IN REQUEST BODY
       msg = parse_body(http_body);
       if(byte_len(msg$text) > 10)
            ++ascore;
       if(match_xss_body in msg$text)
       {
            ++ascore;
            if(match_xss_uri1 in msg$text)
              ++ascore;
       }
       if ( ascore >= 3)
       {
           NOTICE([$note=XSS_Post_Injection_Attack,
                   $conn=c,
                   $msg=fmt("XSS Attack from %s to destination: %s with Attack string %s and post data %s",
c$id$orig_h,       c$id$resp_h, c$http$uri, http_body)]);
       }
   }

}

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 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.

```
# /usr/local/bro/bin/bro -r XSS-dvwa-lowsecurity.pcap xssdetecta2.bro
# cat notice.log
1335932451.038475      RYyyaLw0YT      192.168.149.1   56420   192.168.149.128 80      tcp
HTTP::XSS_Injection_Attack      XSS Attack from 192.168.149.1 to destination: 192.168.149.128 with Attack string
/dvwa/vulnerabilities/xss_r/?name=<script>alert(1)</script> -       192.168.149.1   192.168.149.128 80      -       bro
Notice::ACTION_LOG      6       3600.000000     F       -       -       -       -       -       -       -
```

Figure 16: notice log for DVWA

With WebGoat, the following result was obtained,

```
1339739032.905723      j53yksjOUSc     192.168.245.1   49634   192.168.245.128 80      tcp
HTTP::XSS_Post_Injection_Attack XSS Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack string
/WebGoat/attack?Screen=40&menu=900 and post data
search_name=%3Cscript%3Ealert%280%29%3C%2Fscript%3E&action=FindProfile      -       192.168.245.1
192.168.245.128 80      -       bro     Notice::ACTION_LOG      6       3600.000000     F       -       -       -
      -       -       -       -       -
1339739044.782021      j53yksjOUSc     192.168.245.1   49634   192.168.245.128 80      tcp
HTTP::XSS_Post_Injection_Attack XSS Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack string
/WebGoat/attack?Screen=40&menu=900 and post data search_name=%3Ctest%3E&action=FindProfile      -
192.168.245.1   192.168.245.128 80      -       bro     Notice::ACTION_LOG      6       3600.000000     F       -       -       -
-       -       -       -
    -
```

Figure 17: notice log for WebGoat

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.

```
# Anamoly 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;
   }
   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)
```

Author Name, email@address

```
                ++ascore;
        }


        if ( ascore >= 3)
        {
            NOTICE([$note=SQL_Post_Injection_Attack,
                    $conn=c,
                    $msg=fmt("SQL Attack from %s to destination: %s with Attack string %s and post data %s",
c$id$orig_h, c$id$resp_h, c$http$uri, http_body)]);

        }
    }
}


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,
                    $msg=fmt("SQL Attack from %s to destination: %s with Attack string %s", c$id$orig_h, c$id$resp_h,
c$http$uri)]);

    }

}
```

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.

```
1339916569.727439  YJfjLhYWLA9  192.168.245.1  51245  192.168.245.128 80  tcp
HTTP::SQL_Post_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with
Attack string /WebGoat/attack?Screen=213&menu=1100 and post data
employee_id=112&password=x'or'a'='a&action=Login  -  192.168.245.1  192.168.245.128 80  -  bro
```

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

```
1339916791.542641  q5z39ByqpB1  192.168.245.1  51279  192.168.245.128 80  tcp
HTTP::SQL URI Injection Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli/?id=1'&Submit=Submit  -  192.168.245.1  192.168.245.128 80  -  bro
Notice::ACTION_LOG  6  3600.000000

1339916811.243118  8CtiEfN7jG9  192.168.245.1  51284  192.168.245.128 80  tcp
HTTP::SQL_URI_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli/?id=1'+or+'1'='1&Submit=Submit  -  192.168.245.1  192.168.245.128 80  -
bro  Notice::ACTION_LOG  6  3600.000000  F
```

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

```
1339916791.542641  q5z39ByqpB1  192.168.245.1  51279  192.168.245.128 80  tcp
HTTP::SQL_URI_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli/?id=1'&Submit=Submit  -  192.168.245.1  192.168.245.128 80  -  bro
Notice::ACTION_LOG  6  3600.000000

1339916811.243118  8CtiEfN7jG9  192.168.245.1  51284  192.168.245.128 80  tcp
HTTP::SQL_URI_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli/?id=1'+or+'1'='1&Submit=Submit  -  192.168.245.1  192.168.245.128 80  -
bro  Notice::ACTION_LOG  6  3600.000000  F
```

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

```
1339919061.825546  MgaH1dgw96c  192.168.245.1  51684  192.168.245.128 80  tcp
HTTP::SQL_URI_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli_blind/?id=1&Submit=Submit  -  192.168.245.1  192.168.245.128 80  -  bro
Notice::ACTION_LOG  6  3600.000000

1339919068.746155  MgaH1dgw96c  192.168.245.1  51684  192.168.245.128 80  tcp
HTTP::SQL_URI_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli_blind/?id=1+and+1=2&Submit=Submit  -  192.168.245.1  192.168.245.128 80
-  bro  Notice::ACTION_LOG  6  3600.000000  F  -  -  -  -  -  -  -  -

1339919075.934222  MgaH1dgw96c  192.168.245.1  51684  192.168.245.128 80  tcp
HTTP::SQL_URI_Injection_Attack SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli_blind/?id=5+and+substring(@@version,1,1)=4&Submit=Submit  -
192.168.245.1  192.168.245.128 80  -  bro  Notice::ACTION_LOG  6  3600.000000  F  -  -  -
```

```
-    -    -    -    -
1339919082.299954      MgaH1dgw96c    192.168.245.1   51684   192.168.245.128 80     tcp
HTTP::SQL_URI_Injection_Attack  SQL Attack from 192.168.245.1 to destination: 192.168.245.128 with Attack
string /dvwa/vulnerabilities/sqli_blind/?id=5+and+substring(@@version,1,1)=5&Submit=Submit    -
192.168.245.1   192.168.245.128 80     -     bro     Notice::ACTION_LOG     6      3600.000000     F
```

Figure 22: Notice log for DVWA application (SQL blind 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

Babbin, Jacob. Security Log Management: Identifying Patterns in the Chaos. Rockland, MA: Syngress, 2006. Print.

Bejtlich, Richard. The Tao of Network Security Monitoring: Beyond Intrusion Detection. Boston: Addison-Wesley, 2005. Print.

"Bro Documentation." Bro 2.0 Documentation. N.p., n.d. Web. 16 June 2012. <http://www.bro-ids.org/documentation/index.html>.

"Ryesecurity." : Solving Network Forensic Challenges with Bro. N.p., n.d. Web. 16 June 2012. <http://ryesecurity.blogspot.com.au/2012/04/solving-network-forensic-challenges.html>.

Trost, Ryan. Practical Intrusion Analysis Prevention and Detection for the Twenty-first Century. [United States]: Addison-Wesley Professional, 2009. Print.

Author Name, email@address

Last Updated: June 22nd, 2017

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Cyber Defence Canberra 2017	Canberra, AU	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MDUS	Jun 26, 2017 - Jul 01, 2017	Live Event
SEC564:Red Team Ops	San Diego, CAUS	Jun 29, 2017 - Jun 30, 2017	Live Event
SANS London July 2017	London, GB	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, JP	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS ICS & Energy-Houston 2017	Houston, TXUS	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, SG	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Los Angeles - Long Beach 2017	Long Beach, CAUS	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Munich Summer 2017	Munich, DE	Jul 10, 2017 - Jul 15, 2017	Live Event
SANSFIRE 2017	Washington, DCUS	Jul 22, 2017 - Jul 29, 2017	Live Event
Security Awareness Summit & Training 2017	Nashville, TNUS	Jul 31, 2017 - Aug 09, 2017	Live Event
SANS San Antonio 2017	San Antonio, TXUS	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Hyderabad 2017	Hyderabad, IN	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Boston 2017	Boston, MAUS	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Prague 2017	Prague, CZ	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UTUS	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS New York City 2017	New York City, NYUS	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Chicago 2017	Chicago, ILUS	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Adelaide 2017	Adelaide, AU	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VAUS	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS San Francisco Fall 2017	San Francisco, CAUS	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Tampa - Clearwater 2017	Clearwater, FLUS	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Network Security 2017	Las Vegas, NVUS	Sep 10, 2017 - Sep 17, 2017	Live Event
SANS Dublin 2017	Dublin, IE	Sep 11, 2017 - Sep 16, 2017	Live Event
SANS Paris 2017	OnlineFR	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced