



Interested in learning
more about security?

SANS Institute InfoSec Reading Room

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

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.

Copyright SANS Institute
Author Retains Full Rights

AD



Let Us Hack You.
Before Hackers Do!
It's Here — The CenZic Website HealthCheck

FREE

 **CENZIC**

Request one now 

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."

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).

2. Web Attack Intrusion Detection

Author retains full rights.

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.

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 (Ryeseecurity, 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

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

A simple HTTP analysis script in Bro language is shown below

Author retains full rights.

} }

```
GET /dir/index.html HTTP/1.0
Host: localhost
Authorization: Digest username="admin",
    realm="admin@test.com",
    nonce="deefgghf36594373131",
    uri="/dir/test.html",
    qop=auth,
    nc=00000001,
    cnonce="0e4f323c",
    response="48845fae49393f05355450972504c4abc",
    opaque="48593ehff23336773t" |
```

```
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
  }
}
```



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

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.

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.

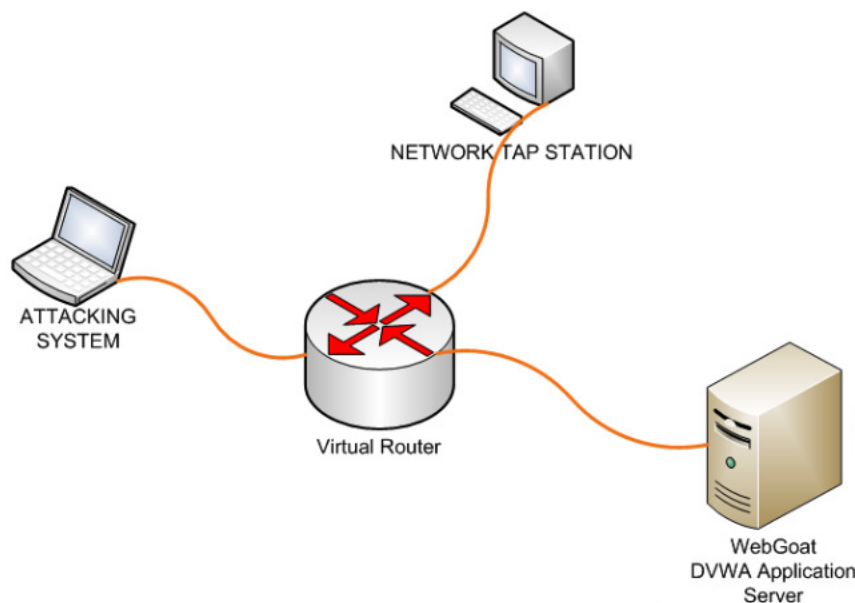


Fig 1a. Test setup.

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	2012-10-10 19:16:20	192.168.2.1	192.168.245.128	HTTP	633	GET /dwa/vulnerabilities/xss_r/?name=0s27Mallory%3C HTTP/1.1
41	2012-10-10 19:16:20	192.168.2.1	192.168.245.1	HTTP	418	HTTP/1.1 200 OK [text/html]
406	2012-10-10 19:16:20	192.168.2.1	192.168.245.128	HTTP	678	GET /dwa/vulnerabilities/xss_r/?name=Hello%27Mallory%25%3Cscript%3Falert%280s29%3C%2Fscript%3E HTTP/1.1
409	2012-10-10 19:16:20	192.168.2.1	192.168.245.1	HTTP	434	HTTP/1.1 200 OK [text/html]
448	2012-10-10 19:16:20	192.168.2.1	192.168.245.128	HTTP	726	GET /dwa/vulnerabilities/xss_r/?name=Hello%27Mallory%3Cscript%3Falert%280s29%3C%2Fscript%3E HTTP/1.1
451	2012-10-10 19:16:20	192.168.2.1	192.168.245.1	HTTP	429	HTTP/1.1 200 OK [text/html]
523	2012-10-10 19:16:20	192.168.2.1	192.168.245.128	HTTP	712	GET /dwa/vulnerabilities/xss_r/?name=0s27Mallory%3Cscript%3Falert%280s29%3C%2Fscript%3E HTTP/1.1
529	2012-10-10 19:16:20	192.168.2.1	192.168.245.1	HTTP	428	HTTP/1.1 200 OK [text/html]
564	2012-10-10 19:16:20	192.168.2.1	192.168.245.128	HTTP	846	GET /dwa/security.php HTTP/1.1
671	2012-10-10 19:16:20	192.168.2.1	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%28%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 07:02:48 GMT

```

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

```

Stream Content
GET /dvwa/vulnerabilities/sqli/?id=1%27or+%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=fg7lm4bp9spbofq129kqnqmr4

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=%27union+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=tk1ke8emvkjnkvddeb3va69u85; acopendivids=phpbb2,redmine;
acgroupswithpersist=nada

HTTP/1.1 200 OK

```

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

47	2012-192.168.1.192.168.149.1	HTTP	462	HTTP/1.1 200 OK (text/html)
105	2012-192.168.1.192.168.149.128	HTTP	605	GET /dwa/vulnerabilities/sql/1?id=27+union+all+select+1%2C%40%40VERSION--+6Submit-Submit HTTP/1.1
108	2012-192.168.1.192.168.149.1	HTTP	534	HTTP/1.1 200 OK (text/html)
155	2012-192.168.1.192.168.149.128	HTTP	679	GET /dwa/vulnerabilities/sql/1?id=27+union+all+select+user%20%29%2Cdatabase%20%29--+6Submit-Submit HTTP/1.1
158	2012-192.168.1.192.168.149.1	HTTP	524	HTTP/1.1 200 OK (text/html)
196	2012-192.168.1.192.168.149.128	HTTP	688	GET /dwa/vulnerabilities/sql/1?id=27+union+all+select+user%2Cpassword+from+users--+6Submit-Submit HTTP/1.1
201	2012-192.168.1.192.168.149.1	HTTP	726	HTTP/1.1 200 OK (text/html)

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

10	2012-192.168.2.192.168.245.128	HTTP	689	GET /dwa/vulnerabilities/sql/1?blind?id=1&Submit-Submit HTTP/1.1
13	2012-192.168.2.192.168.245.1	HTTP	475	HTTP/1.1 200 OK (text/html)
15	2012-fe80::e18 ff02::c	SSDP	208	M-SEARCH * HTTP/1.1
16	2012-fe80::e18 ff02::c	SSDP	208	M-SEARCH * HTTP/1.1
17	2012-192.168.2.192.168.245.128	HTTP	654	GET /dwa/vulnerabilities/sql/1?blind?id=1+and+1%3026Submit-Submit HTTP/1.1
19	2012-192.168.2.192.168.245.1	HTTP	449	HTTP/1.1 200 OK (text/html)
21	2012-fe80::e18 ff02::c	SSDP	208	M-SEARCH * HTTP/1.1
22	2012-fe80::e18 ff02::c	SSDP	208	M-SEARCH * HTTP/1.1
23	2012-192.168.2.192.168.245.128	HTTP	698	GET /dwa/vulnerabilities/sql/1?blind?id=5+and+substring(20%40%40version%2C1%2C1%29%3046Submit-Submit HTTP/1.1
25	2012-192.168.2.192.168.245.1	HTTP	449	HTTP/1.1 200 OK (text/html)
27	2012-fe80::e18 ff02::c	SSDP	208	M-SEARCH * HTTP/1.1
28	2012-fe80::e18 ff02::c	SSDP	208	M-SEARCH * HTTP/1.1
29	2012-192.168.2.192.168.245.128	HTTP	734	GET /dwa/vulnerabilities/sql/1?blind?id=5+and+substring(20%40%40version%2C1%2C1%29%3056Submit-Submit HTTP/1.1
31	2012-192.168.2.192.168.245.1	HTTP	504	HTTP/1.1 200 OK (text/html)

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

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.

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 xIIdz1ltB3 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 - - - - - -
```

```

1339916811.243118      Z0250sTVSGb  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    tBwwYdBF1j  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    ysdIW05nvii 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.

```
## Anomaly 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;
    }
```

```
}
```

```

}

## 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

```

```

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,
           $msg=fmt("XSS Attack from %s to destination: %s with Attack string %s", c$src$orig_h, c$src$resp_h,
c$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  j53yksjOUSe  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%28%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  j53yksjOUSe  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.

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.

```
# 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=/);
```

```

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)

```

```

        ++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$res_h, c$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$res_h,
            c$http$uri)]);
    }
}

```

Figure 18: Bro Script for detecting SQL injection

```
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|||
```

```
1339916791.542641    q5z3BypqB1  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||
```

```

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    8CtEfN7jG9    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

```

```

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    -    -    -

```

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

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, sgul etc which can generate meaningful alerts regarding web attacks.

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.



Upcoming SANS Training

[Click Here for a full list of all Upcoming SANS Events by Location](#)

SANS San Antonio 2013	San Antonio, TXUS	Dec 03, 2013 - Dec 08, 2013	Live Event
SEC480 Beta - Canberra, Australia	Canberra, AU	Dec 11, 2013 - Dec 13, 2013	Live Event
SANS Cyber Defense Initiative 2013	Washington, DCUS	Dec 12, 2013 - Dec 19, 2013	Live Event
SANS Oman 2013	Muscat, OM	Dec 14, 2013 - Dec 19, 2013	Live Event
SANS Golden Gate 2013	San Francisco, CAUS	Dec 16, 2013 - Dec 21, 2013	Live Event
FOR572 Advanced Network Forensics	San Antonio, TXUS	Jan 05, 2014 - Jan 10, 2014	Live Event
FOR585 Adv Smartphone and Mobile Device Forensics	San Antonio, TXUS	Jan 13, 2014 - Jan 18, 2014	Live Event
SANS Security East 2014	New Orleans, LAUS	Jan 20, 2014 - Jan 25, 2014	Live Event
SANS Dubai 2014	Dubai, AE	Jan 25, 2014 - Jan 30, 2014	Live Event
AppSec 2014	Austin, TXUS	Feb 03, 2014 - Feb 08, 2014	Live Event
SANS Cyber Threat Intelligence Summit	Washington, DCUS	Feb 04, 2014 - Feb 11, 2014	Live Event
Secure India@Bangalore 2014	Bangalore, IN	Feb 10, 2014 - Feb 22, 2014	Live Event
SANS Scottsdale 2014	Scottsdale, AZUS	Feb 17, 2014 - Feb 22, 2014	Live Event
FOR508 Tokyo - February 2014	Tokyo, JP	Feb 17, 2014 - Feb 22, 2014	Live Event
SANS Brussels 2014	Brussels, BE	Feb 17, 2014 - Feb 22, 2014	Live Event
RSA Conference 2014	San Francisco, CAUS	Feb 23, 2014 - Feb 24, 2014	Live Event
Asia Pacific ICS Security Summit & Training	OnlineSG	Dec 02, 2013 - Dec 08, 2013	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced