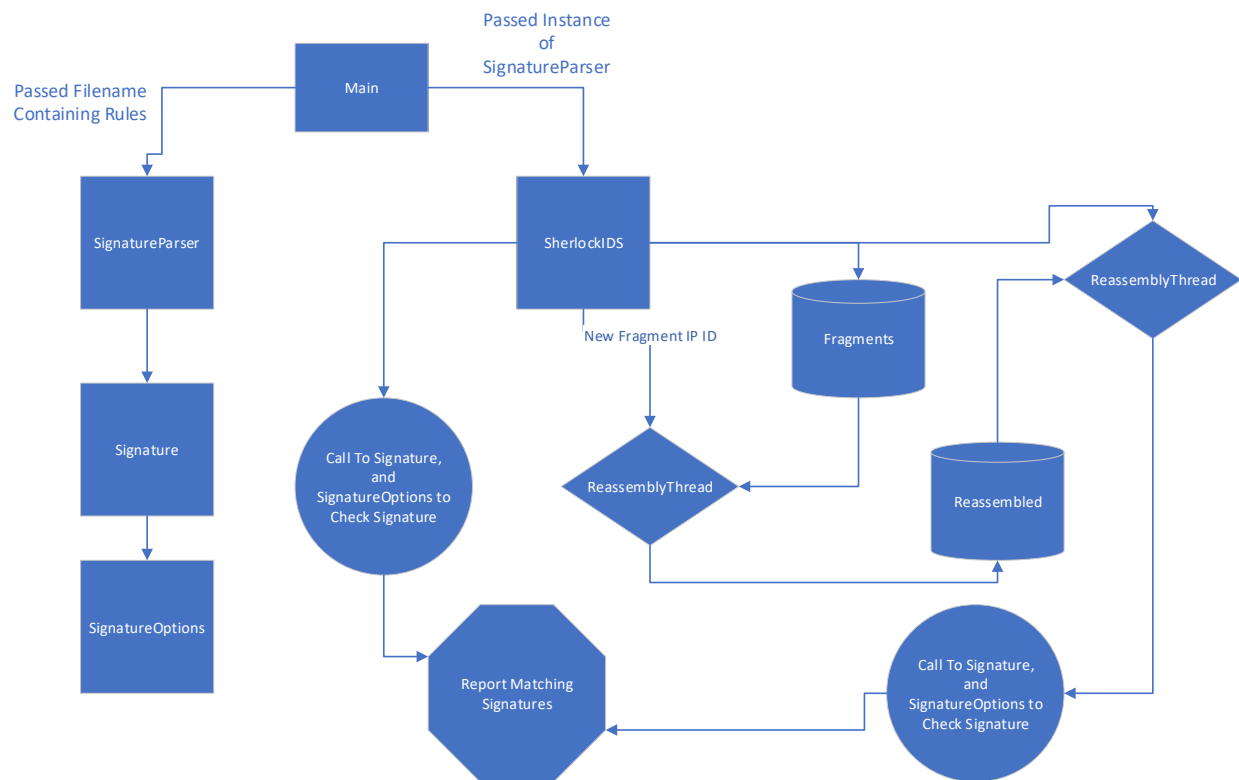


## Intro:

In this project a simple intrusion detection system was built, with the help of functions and classes implemented in the previous two projects. The major addition that was made in this project were three classes: SignatureParser, Signature, and SignatureOptions. As can probably be predicted from the names, these classes handle the parsing of signatures, and in comparing packets to them. The class SherlockIDS was also built, and it would handle the parsing and transferring of packets to fragment handlers, if needed, or to the previously defined classes which would check for a matching signature. The class FragmentAdministrator was created in project 2, to handle the aggregation of reassembled fragment packets. In this project the added functionality of checking whether the reassembled packets matched signatures was added to the FragmentAdministrator. The below diagram shows the infrastructure that was created for project 3.



## Main:

Main for this project served two purposes. The first was to create an instance of the SignatureParser class and pass it the name of the file that contains the rules. The other was to initiate SherlockIDS with the instance of the SignatureParser class it created.

## SignatureParser:

This class starts off by creating a treemap object, the point of this was to have a single object that would contain all the rules in a way that could be retrieved fairly easily. The treemap object had the key of type string, which would be either ip, arp, udp, icmp, or tcp for the protocols accepted, and a value of Vector<Signature> which is the class created that held the signatures. The class then continues to read the rules from the file it was passed, and calls the Signature class to parse the rules, and stores the instance of the class in the treemap object. This class also provides a function that returns the treemap object.

## Signature:

The starting function for this class is called parse. In this function the class receives the rule as a string, parses, and then stores the first seven parts of the rule. These are the action, protocol, ip/mask of source, port or port range of source, direction, ip/mask of target, port or port range of target. It instantiates the class SignatureOptions to handle the parsing and matching of options. The way that the rule is parsed is that the string is split using blank spaces as the delimiter. The array that is received from the splitting of the original rule string is then traversed. The first string is checked to see if it is either "alert" or "pass" and stored. If neither of these values matches, then an error is returned. The second string is checked to see if it is one of the accepted protocols and stored. If it is not, then an error is returned. The third string which is the ip address target and the subnet mask, is split into two strings this is to check that the ip address has four parts to check input. If it does not, then an error is returned. The fourth string which is the port source is split with a colon as a delimiter. If an array two strings is returned, then we know a port range is being defined. This value is checked to make sure that the first is smaller than the second, and if not an error is returned and changed to any as the port. If no range is detected then the defined port is stored as a string, this is in case that any is set.

The fifth string is compared to either the value ->, or <> and Boolean value is set that defines the rule as either bidirectional or unidirectional. If not, then an error is returned. The sixth string which is the ip address and the subnet mask of the target, is split into two strings this is to check that the ip address has four parts to check input. If it does not, then an error is returned. The seventh string which is the port target is split with a colon as a delimiter. If an array two strings is returned, then we know a port range is being defined. This value is checked to make sure that the first is smaller than the second, and if not an error is returned and changed to any as the port. If no range is detected then the defined port is stored as a string, this is in case that any is set. The class then checks if the array is longer than 7, if it is it will pass a substring of the remaining part of the string to SubstringOptions to parse for options.

The function SignatureMatching defined in this class receives an IPPacketParser object which has parsed ip packet as a class to retrieve values, also the source and destination ports of the packet, and a value of portAvailable which tells the function whether the packet has a port. This comes into use when differentiating between icmp packets, and raw ip packets. The

function then checks that the ip source, and destination match the subnet mask value that the rule has. If the rule is bidirectional, then the source and target are switched and checked a second time. The function `printRule` is used to check what values signature has stored. The function `GetProtocol` is used to get the protocol of the rule, this makes it easier to put into the treemap object of the `SignatureParser` class. The `GetSignatureOptions` function retrieves the instance of `SignatureOptions` that this rule has, which contains the rules options, and function used to compare packets to them.

## SignatureOptions:

The starting function for this class is called `parse`, and it receives two strings. The first string is the substring of the original rule that contains the options, the second string is the entire rule, this is to make it easier to debug individual rules. The string containing only the options is then split by using semicolon as the delimiter. Each option is then parsed for the accepted options which are the following:

Option	Description
<b>msg</b>	Prints a message into the log
<b>logto</b>	Log the packet to a user specified filename instead of the standard output file
<b>ttl</b>	Test the IP header's TTL field value
<b>tos</b>	Test the IP header's TOS field value
<b>id</b>	Test the IP header's fragment ID field for a specific value
<b>fragoffset</b>	Test the IP fragment offset against a specific value
<b>fragbits</b>	Test the fragmentation bits of the IP header
<b>flags</b>	Test the TCP flags for certain value
<b>seq</b>	Test the TCP sequence number against a value
<b>ack</b>	Test the TCP acknowledgement field for a specific value
<b>itype</b>	Test the ICMP type field against a specific value
<b>icode</b>	Test the ICMP code field against a specific value
<b>content</b>	Search for a pattern in the packet's payload
<b>sameip</b>	Determines if source IP equals the destination IP
<b>sid</b>	A unique integer identifying the rule

Each option has its own variable with the same name in the class, and is set to the argument of the option, or just to enabled if there is no argument. The function `CheckMatchingTCP` receives the value of a `TCPParser` object, which is a tcp packet. The function will then compare the tcp packet to any of the set options, if it matches all the set options it will return true, otherwise a false. The options it checks for sequence, flags, or acknowledgement.

The tcp flags options requires more processing than some of the other options. The way that it is checked is that a mask is created with the flags that are sent in as an argument. Then when it comes time to compare the tcp flags, the flags of the tcp packet are retrieved and anded

with the masked. According to what the modifier was set with the option different end values are checked. If the modifier + is set, then it checks if the result is equal to the original mask, if it is then there's a match, if not then there is no match. If the modifier \* is set, then it checks if the result is greater than 0, if it is then there's a match, if not then there is no match. If the modifier ! is set, then it checks if the result is equal to 0, if it is then there's a match, if not then there is no match.

The ContentMatching function is used to compare the payload to specific byte values set in the options. It returns true if there is a match in the payload, or false if there was no match. The PayloadSizeMatching function is used to check if the size of the payload matches the one that is set if any. The messageToPrint function returns the message option argument, and the fileToPrintTo function returns the logto option argument. The printOptions function prints the options that are set. The CheckMatchingICMP function receives an argument of an ICMPParser object, which is an icmp packet. The packet is then compared to the icmp options that can be set, these are the itype, and the icode options.

The CheckMatchingIP function receives an argument of an IPPacketParser object, which is an ip packet. The packet is then compared to the ip options that can be set, these are ttl, tos, id, fragoffset, sameip, and fragbits. The fragbits option is like the flags option, and so a similar approach was used where a mask is created based on the fragbits to check for. This mask is then anded, and the result is compared according to what modifiers were set. If the modifier + is set, then it checks if the result is equal to the original mask, if it is then there's a match, if not then there is no match. If the modifier \* is set, then it checks if the result is greater than 0, if it is then there's a match, if not then there is no match. If the modifier ! is set, then it checks if the result is equal to 0, if it is then there's a match, if not then there is no match.

## **SherlockIDS:**

This class is used as the primary driver of the IDS. It receives a treemap object that contains all the signatures. All of the vectors containing the signatures, which are separated by what protocol their checking, are retrieved and stored into their own variables. The SetupInvestigation function sets up where the packets will be coming from. If a file is passed through the command line, then it is used. If no file is set then it detects what network interfaces are available, and then prompts the user to choose one to monitor. The Investigate function is used to start parsing packets. The function will figure out how to parse the packets, if at all, according to the headers. A FragmentAdministrator instance is created is also started in order to handle reassembled packets. If a fragmented ip packet is detected, then a thread is started with the id set in the ip header.

Each ip fragment is then passed through all the ip rules that were set through the function CheckIPRules. This function handles the matching of ip signatures, and if any rules match it will log them to either the log file that is specified in the rules option, or print them to the console. In the fragment scenario is when the fragoffset, and fragbits options are likely to be useful. If a

fragment is not detected, then the ip packet is still checked for any matching ip rules. The protocol of the packet is checked, and the rules of those protocols are checked to see for any matches. If any of the rules contain Sid's to check for then they are ignored, since the fragment administrator is the only one that can check for this. Furthermore, all the types of packets are checked for size and content options. The arp packet rules are checked when the ethernet is set to 0806.

## **FragmentAdministrator:**

This classes main task was to handle the output of the reassembled packets. When this class is started as a thread, it will check if the reassembled packet queue has any items in it. If it does then it will retrieve and remove them. Whenever it retrieves an item, it will record the IP ID of the packet. It will then check if the sid of the reassembled fragment and if its 2 or 4 then it will print either overlap detected for 2 or time out detected for 4. It will then print the reassembled packet.

The secondary task of this class is to flush the queue of any IP ID packets that have already been reassembled. Since it stores all of the IP ID's of the finished packets, it will loop through this list and remove fragments that have already finished.

This class also has the added ability to check the reassembled fragment for signatures. This is done in the same way that the SherlockIDS checks for signatures, except that it does not check for arp packets. The other difference is that SID value is checked. Since different SID values are set according to what happened when the packet was reassembled, then they were checked to any rule that had any SID options.

## **IPFragmentAssembler:**

This class was made to handle the reassembly process of a single IP ID packet fragment. The constructor of this class creates a timestamp of when it was started, which correlates with when the first packet of that IP ID was received. It will also create a treemap object that will hold all its IP fragments. When this class is started as a thread, it will first check if it has successfully reassembled the packet, or it has timed out. If it hasn't then it will check how long it has been since it started by comparing the timestamp to the current time, if more than 10 seconds have passed then it will set the timeout value to exit the loop in the next iteration.

The concurrent queue of IP fragments is checked for the first value. If the first value has the IP ID that the thread is assigned to then it will grab, it and remove. The fragment retrieved is then checked to see if it is either the first or the last. This can be found since the first fragment will have the more fragment flag set, and an offset of 0. While the last fragment will have the more fragment flag unset, with an offset greater than 0. The first and last packets are stored in their own variables, while all the other fragments are stored into a treemap object. The reassembly process will not start until the first and last packet have been received. Once these packets are received then it is checked if they can be put together without any in between

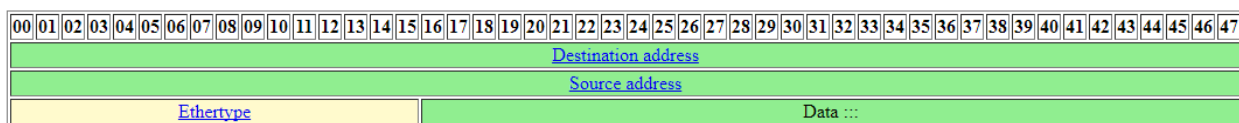
packets, whether there is an overlap or not. If it overlaps it will flag it however. If they can be then they are put together, and the header of the first packet is used.

If the first and last packet can't be put together, then a function called `packetReassembly` is called. This function receives a fragment, a treemap object of fragments, an array of bytes to reassemble the payload, and a value that checks how many bytes the current fragment has. The function will then check to see if it can find a fragment in the treemap object that follows the fragment it received, whether it be overlapping or not. If it overlaps it will flag it however. If it can it will extract the payload of the packets and concatenate them together, it will then call itself recursively. This is done until either a packet can't be found in the treemap object, or the last packet is reached.

When the last packet is reached then it will check if it can put them together, and if it can then it will concatenate the payloads and then return a `CompleteFragment` object. The purpose of this object is to return whether reassembly was a success, and the payload if it was. If the reassembly process was successful then the payload of the fragment is concatenated with the header of the first packet. Also, if at any point the size of the payload exceeds 64k then it is also flagged. The completed packet is then placed into a model called `FragmentModel`. The timestamp, and the treemap of fragments is also stored into the model. Furthermore, if any events such as packet overlapping, or the payload exceeding 64k happen then the appropriate sid is also stored in this model. The sid of overlapping is 2, and the sid of exceeding 64k is 3. If neither of these happened, then an sid of 1 is set meaning the reassembly process was perfect. If while retrieving the packets the thread times out, then a fragment model is again created with the packets received and an sid of 4 which signifies timing out. The fragment model is then placed into the concurrent queue of reassembled fragments, and the thread exits.

## EthernetParser:

### Ethernet Packet



<http://www.networksorcery.com/enp/protocol/IEEE8023.htm>

Ethernet packets are parsed according to the above image. The `EthernetParser` class contains a pair of variables for each different part of the header, except for payload which only has one variable representing it. The first variable stores the value as a byte array, while the second converts it to a string. The conversion to string maintains the value in hex. The payload is stored as a byte array copying the packet that was parsed from index 14 to the end of the array. The class also contains getters to retrieve every part of the header as either bytes, or a string. Furthermore, two functions named `printAll`, and `printHeaderOnly` are defined for printing. The payload is converted into a UTF-8 encoding.

## ARPParser:

### ARP Packet

Internet Protocol (IPv4) over Ethernet ARP packet		
octet offset	0	1
0	Hardware type (HTYPE)	
2	Protocol type (PTYPE)	
4	Hardware address length (HLEN)	Protocol address length (PLEN)
6	Operation (OPER)	
8	Sender hardware address (SHA) (first 2 bytes)	
10	(next 2 bytes)	
12	(last 2 bytes)	
14	Sender protocol address (SPA) (first 2 bytes)	
16	(last 2 bytes)	
18	Target hardware address (THA) (first 2 bytes)	
20	(next 2 bytes)	
22	(last 2 bytes)	
24	Target protocol address (TPA) (first 2 bytes)	
26	(last 2 bytes)	

[https://en.wikipedia.org/wiki/Address\\_Resolution\\_Protocol](https://en.wikipedia.org/wiki/Address_Resolution_Protocol)

ARP packets are parsed according to the above image. The ARPParser class contains a pair of variables for each different part of the header. The first variable stores the value as either a byte array or integer, while the second converts it to a string. All the values except for the MAC addresses are converted into decimal. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default. When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. The class also contains getters to retrieve every part of the header as a string. Furthermore, the functions named printAll is defined for printing.

## IPPacketParsing:

### IP Packet

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP				ECN				Total Length															
4	32	Identification																Flags				Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

[https://en.wikipedia.org/wiki/IP\\_v4](https://en.wikipedia.org/wiki/IP_v4)

IP packets are parsed according to the above image. The IPPacketParser class contains a pair of variables for most of the different parts of the header. The exception is version, ihl, dscp, ecn, and payload. The first four are converted to strings, while payload is kept as a byte array. The first variable stores the value as an integer, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default.

When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. When retrieving values that were less than a byte long, a masking technique was used. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 34 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

## TCPParser:

### TCP Packet

Offsets	Octet	0								1								2								3																							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31																
0	0	Source port																Destination port																															
4	32	Sequence number																																															
8	64	Acknowledgment number (if ACK set)																																															
12	96	Data offset				Reserved 0 0 0			N S	C W R E	E C G	U A K	P S H	R S T	S Y N	F I N	Window Size																																
16	128								Checksum																								Urgent pointer (if URG set)																
20	160	Options (if data offset > 5. Padded at the end with "0" bytes if necessary.)																																															
...	...	...																																															

[https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol)

TCP packets are parsed according to the above image. The TCPParser class contains a pair of variables for most of the different parts of the header, except for the payload, and the flags which are all stored in one variable. The first variable stores the value as either an integer or a long, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default.

When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. When retrieving values that were less than a byte long, a masking technique was used. When parsing flags, the aggregation of all the flags is stored in one variable, the string representation has the different types of flags parsed. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 54 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.



## UDPParser:

### UDP Packet

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

[https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol)

UDP packets are parsed according to the above image. The UDPParser class contains a pair of variables for most of the different parts of the header, except for the payload. The first variable stores the value as an integer, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default.

When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 42 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

## ICMPParser:

### ICMP Packet

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Type								Code								Checksum															
4	32	Rest of Header																															

[https://en.wikipedia.org/wiki/Internet\\_Control\\_Message\\_Protocol](https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol)

ICMP packets are parsed according to the above image. The ICMPParser class contains a pair of variables for most of the different parts of the header, except for the payload. The first variable stores the value as either an integer or a byte array, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default. When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added.

The “Rest of Header” part of the icmp packet is maintained as byte array, and a getter to retrieve it as such is provided. This is to facilitate further processing that this project doesn’t do but might be needed in a later project. When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 42 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

## Testing:

To test the IDS, rules were created to test for the following attacks: winnuke, jolt, and teardrop. These attacks were analyzed from the files given to us and the following rules were made:

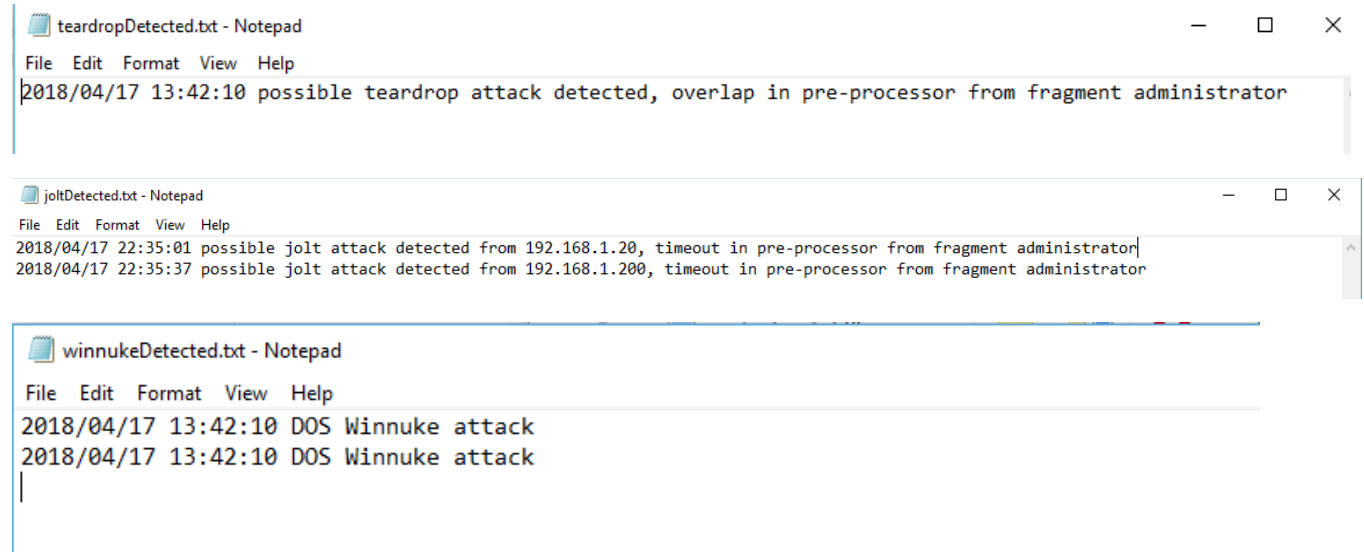
```
alert tcp any any -> 192.168.1.0/24 139 (msg: "DOS Winnuke attack"; flags:U+;  
logto:winnukeDetected.txt;)
```

```
alert ip 192.168.1.200/28 any -> 192.168.1.10/24 any (msg: "possible jolt attack detected from  
192.168.1.200, timeout in pre-processor"; sid: 4; logto: joltDetected.txt;)
```

```
alert ip 192.168.1.20/28 any -> 192.168.1.10/24 any (msg: "possible jolt attack detected from  
192.168.1.20, timeout in pre-processor"; sid: 4; logto: joltDetected.txt;)
```

```
alert udp 129.244.125.194/24 4153 -> 129.244.73.185/24 61045 (msg: "possible teardrop attack  
detected, overlap in pre-processor"; sid:2; logto:teardropDetected.txt;)
```

The results showed success in identifying the attacks according to these rules. The following show the logfiles that were created when the attacks were detected.



The image displays three Notepad windows stacked vertically, each showing log entries from the Sherlock IDS. The top window, titled 'teardropDetected.txt - Notepad', contains a single log entry: '2018/04/17 13:42:10 possible teardrop attack detected, overlap in pre-processor from fragment administrator'. The middle window, titled 'joltDetected.txt - Notepad', contains two log entries: '2018/04/17 22:35:01 possible jolt attack detected from 192.168.1.20, timeout in pre-processor from fragment administrator' and '2018/04/17 22:35:37 possible jolt attack detected from 192.168.1.200, timeout in pre-processor from fragment administrator'. The bottom window, titled 'winnukeDetected.txt - Notepad', contains two log entries: '2018/04/17 13:42:10 DOS Winnuke attack' and '2018/04/17 13:42:10 DOS Winnuke attack'.

```
teardropDetected.txt - Notepad
File Edit Format View Help
2018/04/17 13:42:10 possible teardrop attack detected, overlap in pre-processor from fragment administrator

joltDetected.txt - Notepad
File Edit Format View Help
2018/04/17 22:35:01 possible jolt attack detected from 192.168.1.20, timeout in pre-processor from fragment administrator
2018/04/17 22:35:37 possible jolt attack detected from 192.168.1.200, timeout in pre-processor from fragment administrator

winnukeDetected.txt - Notepad
File Edit Format View Help
2018/04/17 13:42:10 DOS Winnuke attack
2018/04/17 13:42:10 DOS Winnuke attack
```

## **Run/Compile:**

### **Windows:**

```
javac -cp ".;commons-lang3-3.7.jar;commons-net-3.6.jar" -d . Main.java
```

```
java -cp ".;commons-lang3-3.7.jar;commons-net-3.6.jar" Main <filename with rules> <filename  
with packets to analyze>
```

### **Linux:**

```
javac -cp ".:commons-lang3-3.7.jar:commons-net-3.6.jar" -d . Main.java
```

```
java -cp ".:commons-lang3-3.7.jar:commons-net-3.6.jar" Main <filename with rules> <filename  
with packets to analyze>
```