**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 of 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 of 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 SingnatureParser class. The GetSignatureOptions function retrieves the instance of SingnatureOptions 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 |
| msg | Prints a message int the log |
| logto | Log the packet to a user specified filename instead of the standard output file |
| ttl | Test the IP header's TTL field value |
| tos | Test the IP header's TOS field value |
| id | Test the IP header's fragment ID field for a specific value |
| fragoffset | Test the IP fragment offset against a specific value |
| fragbits | Test the fragmentation bits of the IP header |
| flags | Tes the TCP flags for certain value |
| seq | Test the TCP sequence number against a value |
| ack | Test the TCP acknowledgement field for a specific value |
| itype | Test the ICMP type field against a specific value |
| icode | Test the ICMP code field against a specific value |
| content | Search for a patter in the packet's payload |
| sameip | Determines if source IP equals the destination IP |
| sid | 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 then 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 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 compare 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 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 class was modified from the previous project to add the 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.

**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:

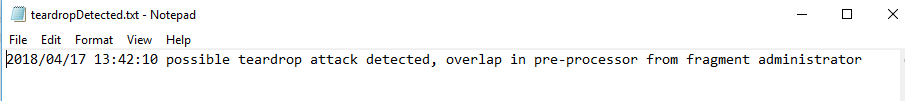
alert tcp any any -> 192.168.1.0/24 139 (msg: "DOS Winnuke attack"; flags:U+; logto:winnukeDetected.txt;)

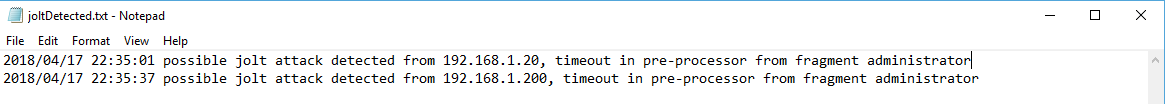
alert ip 192.168.1.200/28 any -> 192.168.1.10/24 any (msg: "possible jolt attack detected from 192.168.1.200, timeout in pre-processor"; sid: 4; logto: joltDetected.txt;)

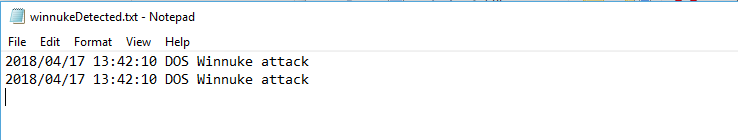
alert ip 192.168.1.20/28 any -> 192.168.1.10/24 any (msg: "possible jolt attack detected from 192.168.1.20, timeout in pre-processor"; sid: 4; logto: joltDetected.txt;)

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.







**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>