CS 487
Homework 5: SNIDS: Simple Network IDS

Revision history

The current version number appears above. Any revisions to this document will be summarized here and the version number above will be incremented. If there is a difference between the one you are using and the one posted here, you are advised to carefully review the changes in revised version.

Ver 1.0: Original version. Ver 1.1: Revisions mainly to update ‘oscar’ to ‘calculus.sisl.rites.uic.edu’

Introduction

This assignment is intended to help you develop an understanding of how packet sniffing and network intrusion detection tools work. It involves developing a simple network intrusion detection system using the libpcap library. It also requires an understanding of how to analyze network traffic, the protocol hierarchy, information in packets and (TCP) sessions.

Note: The coding effort involved in this project is moderate. You can code this project in any language, as long as your final submission can be run according to the syntax described below. Use of the Java programming language is strongly recommended. In this case, you will use the Java pcap library and the `java.util.regex` package. You may have to do a bit of reading before working with the pcap library. Links to these are posted on the class home page.

Logistics

You can do this homework in pairs. Please email the TA the names and email addresses of the people in your group. You can also work on this all by yourself if you prefer.

Submission instructions are described in a later section. Any clarifications and revisions to the assignment will be posted on the course Web page.

Use the discussion board on Piazza to discuss the homework.

Hand Out Instructions

The material needed for this homework is available on the Linux machine `calculus.sisl.rites.uic.edu` under `idslab` subdirectory of the course account (i487). Your code is expected to work on

Start early!!

Start by beginning your working in a (protected) directory. (Note: It is your responsibility to protect your work; Since you are doing a practical course in security, you are expected to keep your work confidential. If you do not know how to set up permissions under Unix, please read Chap 5 of the text book on access controls).

A network IDS

Recall our discussions about firewalls and intrusion detection. Network IDS systems work by looking at live network traffic and filtering or modifying them based on profiles of network behavior and / or specific traffic rules. In this homework, we will almost exclusively focus on *misuse rules*, which will describe BAD network traffic that needs to be filtered. Case in point: The "ILOVEYOU" virus, which arrives in the form of an email payload to your mail server. You will create rules that will filter such virus tcp packets/streams based on the rule database.

To create such a network intrusion detection system, you'll need access to a machine with superuser privileges that will allow you to sniff and filter traffic. To avoid this problem, in this homework you will work on network traces. It should relatively be a straightforward and simple procedure to adapt the system you build to real network traffic on your own machine.

You must note that building the framework is only the first step in detecting intrusions. The real issues that concern IDS systems are false positives (fake alarms), reliability (accurate matching and false negative elimination) and performance. (Of course, we do not have the time to look at these issues, in this homework).

Specifically you will create a executable called `snids` that will take two arguments (executed as `snids <arg1> <arg2>`. The first argument is be a rule file whose syntax is described below. The second argument is the network trace file.

The expected output of the program will be to just print the name of the rule (as described below) to STDOUT.

A rule file must have one (and only one) host entry which denotes the host that the IDS is running on and arbitrarily many rule entries.

```
           <host> ::= host=<ip>\n\n
           <rule> ::= name=<string>\n
                      <tcp_stream_rule>|<tcp_protocol_rule>\n
<tcp_stream_rule> ::= type=stream\n
                      local_port=(any|<port>)\n
                      remote_port=(any|<port>)\n
                      ip=(any|<ip>)\n
```

```
                       (send|recv)=<regexp>\n
   <tcp_protocol_rule> ::= type=protocol\n
                       proto=tcp|udp\n
                       local_port=(any|<port>)\n
                       remote_port=(any|<port>)\n
                       ip=(any|<ip>)\n
                       <sub_rule>
                      <sub_rule>*
          <sub_rule> ::= (send|recv)=<regexp> (with flags=<flags>)?\n

            <string> ::= alpha-numeric string
                <ip> ::= string of form [0-255].[0-255].[0-255].[0-255]
              <port> ::= string of form [0-65535]
            <regexp> ::= Perl Regular Expression
             <flags> ::= <flag>*
              <flag> ::= S|A|F|R|P|U
```

`ip, remote_port, recv` all refer to the remote site of the network connection. Also a ⋆ denotes zero or more repetitions and a `?` denotes zero or one repetitions. TCP flag can be one of six values: a) F : FIN - Finish b) S : SYN - Synchronize; c) R : RST - Reset 4) P : PUSH - Push 5) A : ACK - Acknowledgement 6) U : URG - Urgent.

A rule is one of two types: stream rules and protocol rules. Stream rules require you to reconstruct an entire (send or receive) TCP stream and then apply filter expression to the entire stream. A TCP stream is a sequence of (send or recv) packets that start with SYN and end with FIN with all sequence numbers in order and filled. Richard Stevens' TCP/IP Illustrated and Unix Network Programming are good references if you are unfamiliar with how TCP reconstruction works.

Protocol rules require you to apply the rule on (sequences of) single packets and have subrules that match a protocols flags, destination address and such with constant values. The subrules must be matched in the order specified in the rule database (see the POP example below).

For example the following rule looks for the "I love you" virus in email payload.

```
host=192.168.1.1
name=I Love You
type=protocol
proto=tcp
local_port=25
remote_port=any
ip=any
recv="ILOVEYOU"
```

In this case, the output of your IDS will be just "I Love You" (rule name) to STDOUT. (Do not print any other message other than the rule name specified in the rule file. Our auto grader will be just doing a `diff` output against expected output and will fail if you print extraneous messages.) Real IDS systems can do much more (such as modifying packets), but for the purposes of this homework, we will just restrict ourselves to printing the matching rules.

A packet may match several rules. In this case, you must print all the rules that match. There is no specified order for printing these matched rules. It can be arbitrary, depending on your implementation.

Another example looks for the same virus in a stream (longer email message).

```
host=192.168.1.1
name=I Love You Stream
type=stream
local_port=25
remote_port=any
ip=any
recv="ILOVEYOU"
```

Another example involves matching all plaintext POP login sessions to a mail server.

```
host=192.168.1.1
name=Plaintext POP
type=protocol
proto=tcp
local_port=110
remote_port=any
ip=any
send="\+OK.*\r\n"
recv="USER .*\r\n"
send="\+OK.*\r\n"
recv="PASS.*\r\n"
send="\+OK.*\r\n"
```

This is an example of a protocol rule, where you are trying to match a set of packets that are *in order*. In the above example, you are looking for a sequence of packets that correspond to an insecure login in a POP session where the user uses plaintext password logins.

In short, TCP Stream requires you to ”rebuild” content divided into multiple packets in a TCP connection and then check the regular expression. Protocol rules start from the assumption that the content of every packet is not divided in chunks.

I have made some test cases available in the same ”idslab” directory on calculus. Look at the file `rules.txt` for some sample rules that correspond to these test cases.

Regular Expressions

Regular expressions are a standard tool used for pattern matching; in this project, they will be used to specify patterns either in individual protocols or in reconstructed TCP streams which should trigger an alert. Java provides several classes for dealing with regular expressions; an introduction can be found in the Java API entry for the Pattern class. You will need to import the package java.util.regex in order to use the Pattern and Matcher classes. Note that, unlike the examples given in the API entry for Pattern, you may find the method Matcher.find to be more useful than Matcher.matches.

If you are having difficulty understanding regular expressions, you might try reading some of the many tutorials available on the web. Although Java handles regular expressions through objects and methods, the actual regular expression syntax is nearly identical to that used in scripting languages like Perl and Python.

Other tools

You can use the tool `ethereal` or `Wireshark` to construct your own traces. Of course, this requires a machine with root access. On the other hand, I will post some traces that you can use for this homework. The `Netdude` tool can be used to edit tcpdump files. Again, these are not required, you can do the homework with the material supplied.

Hints and suggestions

.

1. The grammar has been designed to avoid using Lex/Yacc. You can write a simple parser in C or Java. Do not worry about parser error handling or invalid inputs to the parser. A set of simple tests for your parser are the above rule examples. See if your parser handles the above examples correctly (you can do this by printing the parsed input). If you still think you'll need help with writing a parser, ask me for some suggestions.
2. Ignore any errors out of reconstruction (with missed packets etc). Make your own decisions about what do in such cases.
3. Your implementation should handle any packets that arrive out-of-order.
4. The PERL regular expressions have been included to make the job of searching for hex values easier in the payload. Very often, you'll encounter packets (esp. with attack strings that contain instructions very similar to your buffer overflow homework) with hex instructions.
5. Break your job into smaller chunks. Write the parser function before moving on to actual pcap filter implementation.
6. Read the Pcap documentation and API thoroughly to get ideas about how to proceed with the implementation.
7. Using the WireShark tool (www.wireshark.com) may be useful for visually inspecting the network trace files.

8. There are many corner cases to consider for this project, and the details of exactly what functionality you should provide are up to some interpretation. When you encounter such a situation, make sure you explain the choice your group made in your essay documentation. (For instance, in reality, there be a chance that there are no FIN packets for a stream. For your implementation, you can assume that a stream is complete with FIN packets.)

Submission Instructions

The following files must be submitted. Use Blackboard to submit the homework.

- Your implementation files.
- A Makefile – This should compile all the classes in your implementation. We will only run the command `make` to see if your programs compile. At the minimal case, your programs should compile for you to get some credit.
- If you prepared any extra test cases, include this in your submission.
- A `Readme` file that should give us any reasonable special instructions that need to be adhered to run your code.
- A short essay from each of your team members describing your experiences with building this IDS. In particular, I would like to know what you think would be the issues (false positives, dependability and performance). Also, list all sources (human, literature, Internet sites) consulted. Note: Each team member must provide an independently authored essay.

Java implementation

Doing this homework in Java will save you from the trouble of dealing with segfaults. Also, some of the object oriented features come handy in such an implementation.

Read the following text in this section, `only` after you have read the jpcap library documentation.

In order to use the jpcap library, you will need to add the file jpcap.jar to your CLASSPATH and the directory containing the file libjpcap.so to your LD_LIBRARY_PATH. (Both these files are available in the project directory). If you are using bash, you can accomplish this with the export command, as in:

```
$ export CLASSPATH=/home/i487/idslab/java-impl/jpcap.jar:
                                        $CLASSPATH
$ export LD_LIBRARY_PATH=/home/i487/idslab/java-impl/jpcap.jar:
  $LD_LIBRARY_PATH
```

In order to process a packet capture file, you will want to create a `packetcapture` object and initialize it with the `openoffline` method. You can then call the addPacketListener method to enable a listener you've created - which will perform the work of your intrusion detection system - and finally call capture with a negative argument. Your program will then process packets until a CapturePacketException is thrown; you should catch this exception and terminate gracefully, as it most likely indicates the end of your capture file.

The addPacketListener method will accept any object that implements the PacketListener interface. Your listener class - which may be declared as, for example, class IDSListener implements PacketListener - must provide a method packetArrived with return type void and argument type Packet. This method will be called on each packet in the capture file (in the order in which they appear); from there you can deal with the packet in whatever way you choose.

You may also use the setFilter method to place a global filter on the packets you examine; any packets that do not match this filter will be ignored, and your packetArrived method will never be called on them. You can find the syntax of filter expressions by looking at the tcpdump man page under ”expressions”. Note that you do not need to use this method, as there is nothing wrong with having your PacketListener examine every packet, so if you're having difficulty constructing a filter that will match all the packets you're interested in, feel free to ignore this feature of jpcap.

Traditionally, network programmers have had to look carefully at flags and match against special constants to separate out different varieties of packets. Luckily for you, the jpcap library puts this distinction into Java's type system. The type hierarchy under Packet reflects the various different protocols used on the Internet; this allows you to use Java's instanceof operator to determine whether an IP packet (of type IPPacket) is in fact a TCP packet (of type TCPPacket). Once you have made this distinction, we highly recommend that you cast the packet to its more specific type and pass it to a method designed for that packet type. You can safely ignore any non-Ethernet packets you encounter; in fact, the current implementation of jpcap has no way of producing any such packets.

All of the classes mentioned above live in the package net.sourceforge.jpcap.capture, which you will probably want to import (import net.sourceforge.jpcap.capture.*;) at the top of any file that refers to them. Other classes in jpcap may live in other packages; details are given in the documentation.

Each packet type defines new methods for giving you information on that variety of packet. Make use of these whenever possible; trying to interpret raw header data directly is much more likely to lead to subtle bugs. All of the packet types also include toString, toColoredString, and (sometimes) toColoredVerboseString methods; these should be very useful in debugging.

In order to match packet data against a regular expression, you will need to convert the packet data (a byte array) to a string. The class String has a constructor that takes a byte array, which should do what you want. If it does not seem to be behaving properly - which may be the case if you are using non-standard locale settings - there is also a constructor that takes, as a second argument, a character set name; try "ISO-8859-1" if you want to prevent any unintended processing of special characters. You may also use the constant 0 as the second argument in order to force correct behavior; this constructor is deprecated, but it is acceptable for use in this project.

C Implementation

You should really do this homework in Java. However, if wish to do this in C, you should use PCRE to match perl style regular expressions on calculus. A POSIX style (regexec) option is also available if you use the `-lpcreposix` flag and the `pcreposix.h` header file. The libraries are available in `/mnt_nfs/home4/instruct/i487/pcre-6.3/lib` and the header file is available in the `/mnt_nfs/home4/instruct/i487/pcre-6.3/include` directory. The man pages are in `http://www.pcre.org/pcre.txt`. You'll need to scroll down (search for PCREPOSIX).

Pay attention to host and network byte orders when processing packets.

Copyright