Programming Assignment #3—Intrusion Detection

CptS 427/527—Computer Security
Assigned: 13 November 2013,
Revised: 25 November 2013
Due: 6 December 2013, 11:59pm

Objective:

Implement a very simple intrusion detection system that uses misuse, anomaly detection, and specification modeling algorithms.

Overview:

Design and implement programs to send and receive messages via an Intrusion Detection System. Alice will send messages to Bob via Irene (the IDS). Rather than implement a complicated protocol, Alice reads lines of text from the input file, one by one, and sends them to Bob (via Irene). Each line of text is treated as a unique message and therefore sent and received individually. This means that messages never contain more than two lines of text. Messages should be checked for misuse signatures (e.g., bad keyword, excessive length), anomalous behavior (e.g., message that are either too short or too long), or control messages that do not follow the specified sequencing.

Software Design & Implementation Tasks (75 pts)

1) Alice – alice (5 pts)

Purpose: Implement an application that reads a text file and sends its contents line by line (i.e., message by message) across a network to a receiving application.
Mandatory Input: a text file.
Mandatory Outputs: none.
Optional Outputs: input may be echoed to the screen (i.e., `stdout`) when it is transmitted.
Command-line argument: an input file (-i *input_file)* and the hostname (-h *hostname*) and port (-p *port*) of the receiving application.

2) Bob – bob (5 pts)

Purpose: Implement an application that receives lines of text from a network socket and writes them to an output file.
Mandatory Input: none.
Mandatory Outputs: a log of the text messages received.
Optional Outputs: received messages may be echoed to the screen (i.e., `stdout`) as they are received.
Command-line argument: an output file (-o *output_file)* and the port (-p *port*) upon which to listen for incoming messages.

3) Irene – irene (5 pts)

Purpose: Implement an IDS that scans all message being sent from Alice to Bob. Messages that violate security policies must be dropped. Additional details are provided in Sections 3b-3d.
Mandatory Input: none.

Mandatory Outputs: an audit log of received messages and associated IDS actions. The audit log shall be in a three-column format. Column 1 is for the message count, column 2 is for the IDS disposition (i.e., "FORWRD", "MISUSE", "ANOMLY", or "SPECIF"), and column 3 contains the received message (no matter how long or short it is).
Optional Outputs: audit log messages may be echoed to the screen (i.e., `stdout`).
Command-line argument: a source port (-s *source_port)* and the destination hostname (-h *hostname*) and port (-p *port*) of the receiving application as well as the audit log file (-l *log_file*). One IDS misuse keyword (–m *misuse_keyword*) and a misuse threshold (-t *misuse_threshold*) are also specified on the command line.

3a) Audit Log (15 pts)

Your IDS should log data about the messages that it receives and whether these messages were routed on to their intended receiver or simply dropped.

Minimum Requirements: Demonstrate that your IDS can record when messages are received and whether these messages were forwarded on or dropped/deleted by the IDS.

3b) Misuse Detection (15 pts)

Design and implement two misuse detection rules, one for matching the keyword signature and the other for messages that are too long. Misuse messages shall be logged and dropped. The misuse rules shall never flag, log, or drop control messages for misuse.

3c) Anomaly Detection (15 pts)

Design and implement an anomaly detection algorithm based upon message length. Message lengths are not expected to vary by more than 10% from the *current* average. Messages with anomalous lengths shall be logged and dropped. Your IDS shall update its current average message length *after* every *valid* message. The *current* average shall be based upon the last ten *valid* (i.e., passed all IDS tests), non-control messages received.

Note: The first message (i.e., first line of text from the input file) can never have an anomalous length because at that point in time the average message length is undefined and therefore the first message must, by definition, be average given that there no prior messages have been received. However, when the second valid message is received, a valid average message length exists, which by definition must be the length of the first valid message. After two valid messages have been sent and received, the average message length will be based upon the length of these first two messages. This continues until ten valid messages have been sent and received. After which, only the last ten valid message lengths are used to compute the *current* average message length. (Assuming of course, that the input lines pass all IDS tests and are not control messages).

3d) Specification Modeling Detection (15 pts)

Design and implement specification modeling detection algorithm that ensures that "control" message are sent in the proper order per the following specification. Four control signals exist, they are: `<READY>`, `<SET>`, `<GO>`, and `<RESET>`. Valid control messages strictly follow the format "`**<CTRL>**`" and occupy a complete line of text by themselves. Thus "`**READY**`" and "`**GO**`" are valid control messages, whereas "`**GO**123`" (extra characters at the end of the line), "`almost**READY**`" (extra characters at the beginning of

the line), and "**READ****SET****GO**" (multiple commands on one line) are not valid control messages and should therefore be processed as regular messages. Control messages shall only be issued in order from <READY> to <SET> to <GO>, after which the commands may be repeated. Additionally, the <RESET> command can be used to abort/reset the cycle back to an initial state where the next expected command would be <READY>. Any commands issued out of order violate this specification and should therefore be both logged and dropped.

Documentation Tasks (25 pts)

1) Cryptographic Checksum of your Source Code (0 pts)

Compute an md5 or SHA1 checksum of a tar file (or similar archive) of your source code and executable image. Include the name of this file, it's size, and it's checksum in your write-up. Assignments turned in without a checksum will be subject to a three point grading penalty.

2) Abbreviated Software Design Document (10 pts)

Briefly describe the data structures and algorithms used to implement the three required tools. Give the sources of all third-party code and cryptography tools/mechanisms that you used in this assignment. (1-2 page limit)

3) Software Test Document (10 pts)

Describe your software test plan and methodology that you used to verify that you implemented the three required software tools correctly. List any know deficiencies. (1-2 page limit).

4) Lessons Learned / Project Evaluation (5 pts)

Discuss the lessons learned from this project. Are there improvements that should be made in a future release? Other possible questions to address: Was this assignment as easy as you anticipated? Did implementing your IDS inspire you to become a network security analyst? Etc.

5) Time Logs (Optional, no points)

List the time spent during software design, implementation, testing. Also list the time spent writing up your project.

Automated Grading (0 pts)

1) Source code tarball (0 pts / 2 penalty pts)

Create a gzip'd tar file containing the source code for the three programs specified within this assignment. Executing the command: `tar -xzf cpts427p2LAST_NAME.tar.gz` shall extract your source code and a Makefile in to the ***current*** working directory.

2) Makefile / Building your programs (0 pts / 3 penalty pts)

Executing `make` shall build/compile the three programs specified within this assignment. If your programs do not need to be compiled (e.g., you are using an interpreted scripting language), then your Makefile's 'all:' target may be empty. *Programs do not compile/build with your 'Makefile' will not receive full credit.*

3) Automated test script (0 pts / 5 penalty pts, *with additional grading impacts***)**

An automated test script will be used to grade your programs within a VirtualBox Ubuntu VM image. Example scripts will be provided at http://www.tricity.wsu.edu/~mckinnon/cpts427/prog3/. A demo script will be used to extract

your gzip'd tar file into a subdirectory, run make, and then it will execute the test script. The 32-bit Ubuntu VM image that was provided for Program#2 (cpts427-fall2013b-vm.ova) contains the required CptS427 and CptS527 accounts. Your programs will be graded within this 32-bit Ubuntu VM image. *Programs that do not execute with the test script will not receive full credit. This means that the actual value of your programs executing as indicated in the test script is worth much more than 5 pts!*

Submission

Email Drs. McKinnon and Manz a PDF file containing your software documentation and a gzip'd tar file containing the source code for the three programs specified within this assignment. The email message shall conform to these requirements:

subject line: `cpts427: Program#3 from LAST_NAME`
writeup file: `cpts427p3LAST_NAME_writeup.pdf`
tarball file: `cpts427p3LAST_NAME.tar.gz`

Additional Details

- You may implement this assignment in any programming languate that is supported by the provided Ubuntu VirtualBox VM image (e.g., C/C++, Java, Perl, Python).
- At least one week before the assignment is due, a set of test files and test script(s) will be provided at http://www.tricity.wsu.edu/~mckinnon/cpts427/prog3/. You will not receive full credit if your tools cannot correctly process these test files. The example test files and scripts will not be sufficient to *fully* test your tools—this is why the write-up contains a test plan section.
- All grading will occur on the provided VirtualBox Ubuntu VM image that was provided for Program#2 (cpts427-fall2013b-vm.ova).
- Programs that do not execute with the provided demo/grading script(s) and virtual machine *will not receive points for correct execution*.