SSIE - SUJET DM

EPITA S8 - 2024

PRIAMOS

Cabinet de conseil et d'expertise en sécurité de l'information

Sujet DM – EPITA S8 - 2024

[Page vierge]

PRIAMOS – SARL unipersonnelle au capital de 21000 € enregistrée au RCS de Toulouse
SIRET : 879959252 00021 | TVA : FR70879959252 | APE : 7022 Z | Organisme de formation n°76310985331
240 rue Jean Bart, 31290 Labège, France
+33 (0)6 46 91 92 13 | contact@priamos.fr | www.priamos.fr | www.linkedin.com/company/priamos/

A Devoir Maison

Ce devoir maison est basé sur l'environnement et les sujets Labtainers.

Le fonctionnement pour télécharger l'environnement et les labs correspondants est le même qu'en TP. Sujets

Deux sujets seront étudiés pour ce devoir : **macs-hash** et **pcap-lib**. Les énoncés sont mis à disposition avec le sujet.

Pour le sujet pcap-lib, vous répondrez à **l'ensemble des questions du sujet. Attention, les temps d'exécution des scripts peuvent être longs, tenez en compte (ne faites pas le DM au dernier moment).**

Pour le sujet macs-hash, vous répondrez **aux questions 1 à 14.**

A.1 Consignes de rédaction

Votre rapport devra être rédigé de manière détaillée, explicite et complète. Toutes les questions devront être adressées.

Le rapport sera rédigé en français, même si les questions du sujet en anglais peuvent être reprises telles quelles.

Des captures d'écrans sont attendues pour justifier vos réponses et votre raisonnement.

A.2 Modalités de rendu

Un rapport par personne sera rendu pour le 21/06/2024 à 18h00. Aucun retard ne sera accepté.

Note : Un rapport par personne i.e. un travail individuel.

Le rapport devra être envoyé par mail à etienne.gerain@priamos.fr, eva.soussi@priamos.fr et epita-ssie@priamos.fr.

PCAP Library Programming

1 Overview

The pcap (packet capture) format is a standard and portable representation of packet-level network traffic. The pcap library (http://www.tcpdump.org/) provides many routines1 to interface with both live (off the network card) and stored (previously captured) network traffic. You are already familiar with pcap from packet introspection lab both Wireshark and tcpdump store and read data in pcap format. This lab is designed to familiarize students with the pcap format and library as the basis for performing arbitrarily complex network traffic analysis tasks, especially across large data sets. For example, one might:

- Compute the fraction of web traffic on a link
- Measure the rate of traffic to a particular destination
- Discover anomalous packets
- Find scanning worm traffic
- Etc, etc.

by analyzing packet traces. This lab begins by investigating the pcap format and library/API. We assume basic familiarity with a UNIX environment, including programming tools. As a refresher, a reasonable UNIX tutorial is available at: http://www2.ocean.washington.edu/unix.tutorial.html. Remember, man pages are your friend[1]. Future traffic analysis labs build on the concepts here it is imperative that you understand these basics in order to be successful with subsequent lab work. There are 16 questions in the lab. Submit your written lab report, including your program code, to your instructor, e.g., via a CLE.

This lab includes software development in a UNIX environment. There exist pcap libraries for all major programming languages; developers of this lab actively use C++ and Python, and have included sample tutorial code for each. Many UNIX distributions include the pcap library and headers by default. Python users are recommended to use the dpkt library. dpkt is readily available via the package manager for most distributions of Linux, and is already installed on this Labtainer. There are a number of resources for understanding the dpkt API; it is recommended you start by reviewing the python tutorial code supplied in the home directory of this lab's computer. There also exist a number of online resources for API documentation. [2]

The language for completing this lab is up to you, however it is important that you become comfortable writing libpcap programs and are capable of running UNIX-style scripts and tools. Note that a C++ program may be significanly slower than a Pthon program, depending on what kinds of data structures you employ.

You may find the programs that you write for this lab are useful for future labs (or other exploration). You are encouraged to place your code and scripts in the `mystuff` directory as described below in 2 so that it persists independent of this lab.

1.1 Background

The student is expected to have an understanding of the Linux command line, and software development tools, e.g., Python or C++. The lab requires plotting of data, and thus some experience with with the python numpy and matplot packages will be helpful [3].

[1] man pcap

[2] `http://dpkt.readthedocs.org/en/latest/api/api_auto.html#module-dpkt.pcap`

[3] `https://matplotlib.org/tutorials/introductory/pyplot.html`

2 Lab Environment

This lab runs in the Labtainer framework, available at http://nps.edu/web/c3o/labtainers. That site includes links to a pre-built virtual machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers.

From your labtainer-student directory start the lab using:

```
labtainer pcap-lib
```

It may take a while for the lab to start, as you will see, it contains a very large pcap file. A link to this lab manual will be displayed.

The home directory of the resulting computer contains a directory named `mystuff`. That directory is shared with your Labtainers host, at:

```
labtainer-student/mystuff
```

Files and directories that you create in `mystuff` will persist independent of this lab (and other labs that make the mystuff directory available). Consider placing your code and scripts there, but do not move the pcap file.

3 Tasks

This lab includes some detailed instructions to facilitate grading. Failure to follow those instructions may result in your lab results not being properly recorded.

3.1 Unknown trace

We will be analyzing an anonymized packet trace taken from an Internet exchange point. This trace is in your home directory in: trace2.pcap. **Do not move this file.**

- Try viewing the trace in either tcpdump or Wireshark. Clearly, (if it opens at all) the packet trace is much too large for a human to make sense of it, draw conclusions, or understand the traffic statistics. We will write a program using a pcap library to analyze the trace. You can use either Python or C++ to complete this lab. The appropriate pcap manipulation libraries for both languages are already installed on the VM. Relevant libraries and documentation links:
- C++: #include <pcap/pcap.h> (http://www.tcpdump.org/manpages/pcap.3pcap.html)
- Python: import dpkt (https://dpkt.readthedocs.io/en/latest/api/api_auto.html#module-dpkt.pcap)

Start by answering the following questions:

1. [5 pts] What link-layer is included in the trace?
2. [5 pts] What is the snap length and what is the significance of the snapshot length? The link type defined in the packet trace header is important as we must skip over the correct amount of data to reach the IP packet (which is what were really interested in). Note that while pcap is the most popular and widely accepted packet capture format, it has several limitations, which have led to development of alternatives. For example, PcapNg, or next-generation pcap, is now the default format in Wireshark.
3. [5 pts] Find the documentation for PcapNg online. Briefly (no more than 2 or 3 sentences) describe the differences between pcap and PcapNg.

3.2 Basic traffic stats

Add to your pcap analysis program the ability to loop through all of the packets of the trace. Each packet in a pcap trace is preceded by a header as defined in pcap pkthdr. This header contains a UNIX timestamp, among other fields. To iterate through all packets of the trace, you may wish to use, in C, pcap loop() or pcap dispatch() with the appropriate callback (you must create the callback). With python's dpkt, `pcap.Reader` provides a similar iterator. Note you may wish to place a bound on the number of packets processed while you are developing your program – but do not forget to remove this bound to complete the assignment.

When answering the following questions, your program **must** output the desired value to stdout with the specified prefix (when present). For example, if the qustion is:

1. [5 pts] How many IPv4 packets does the trace contain (as IPv4 count:)?

then, your program should output a line containing:

```
Count IPv4:  343
```

or whatever the value is. Unless otherwise directed, output values as an integer with no formatting, e.g., no thousands commas. The order in which your program outputs the answers does not matter. Nor does it matter if different programs output results of different questions.

1. [5 pts] How many IPv4 packets does the trace contain (as IPv4 count:)?
2. [5 pts] How many non-IPv4 packets does the trace contain (as non-IPv4 count:)?
3. [5 pts] What is the timestamp of the first packet in the trace, including at least two decimal places. (as First timestamp:)?
4. [5 pts] What is the average packet rate (in packets per second to two decimal places) of the trace (as Avg packet rate:)?

After the per-packet pcap header is the traffic data. Improve your callback to decode IP packets. Your callback should obtain the source and destination IP addresses, the protocol (e.g. TCP, UDP, ICMP, etc), source and destination transport port (where applicable), etc. Several questions ask for an answer in the form of a distribution. Please note that a distribution must represent each value in the form of a fraction or percentage of the total. Answer the following questions:

5. [10 pts] What is the packet protocol distribution? (A table showing the 5 top protocols and their respective contributions is fine.)
6. [10 pts] Plot a histogram of the packet size distribution (the Python numpy and matplot packages are installed on the Labtainer).
7. [5 pts] How many unique IPv4 source addresses are present in the trace (as Unique sources:)?
8. [5 pts] How many unique IPv4 destination addresses are present in the trace (as Unique destinations:)?
9. [10 pts] Create a cumulative distribution function (CDF) plot. The x-axis is the number of bytes sent and the y-axis is the cumulative fraction of sources.
10. [5 pts] Which source sent the most bytes (as Source with most bytes:)?
11. [5 pts] Which source sent the most packets (as Source with most packets:)? Based on your analysis of the trace:

12. [10 pts] List 3 characteristics of the traffic that seem unusual to you.

13. [5 pts] Provide a reasonable explanation for what traffic the trace represents, taking into account the unusual characteristics you have identified.

4 Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab
```

When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.

This lab was originally created by faculty at the Naval Postgraduate School Department of Computer Science. It was adapted for the Labtainer framework by the NPS Center for Cybersecurity and Cyber Operations under sponsorship from the DoD CySP program.

Exploring MACs and Hash Functions

Getting Started

Boot your Linux system or VM. If necessary, log in and then open a terminal window and cd to the labtainer/labtainer-student directory. The pre-packaged Labtainer VM will start with such a terminal open for you. Then start the lab:

```
labtainer macs-hash
```

Note the terminal displays the paths to three files on your Linux host:

1) This lab manual
2) The lab report template
3) A spreadsheet that you will populate as part of the lab.

On most Linux systems, these are links that you can right click on and select "Open Link". **If you chose to edit the lab report and/or spreadsheet on a different system, you are responsible for copying the completed files back to the displayed path on your Linux system before using "stoplab" to stop the lab for the last time.**

Familiarize yourself with questions in the lab report template before you start.

In this lab, you will explore cryptographic hash functions and message authentication codes using `openssl`, `shasum`, and a couple of home-grown scripts.

Note: There is an appendix with commonly used Unix commands.

Use the "ll" command to list the content of the directory.

Task 1: Practice Generating Digests

For this task you will **not** be using OpenSSL for digest creation because there are easier ways of generating digests on a Unix system. Instead, you will be using the `shasum` command, which by default supports SHA-1 (with a 160-bit output), as well as other SHA-2 outputs as an option. You can see the options by entering the following[1]:

```
shasum --help | less
```

The `shasum` command uses the following syntax:

```
shasum –a ALGORITHM FILENAME
```

replacing:

- *ALGORITHM* with one of the supported options (i.e., **1, 224, 256, 384, 512, 512224, 512256**).
- *FILENAME* with the name of some file on your system.

For example, to generate the 160-bit SHA-1 digest:

```
shasum –a 1 foo.txt
```

Do the following:

1. Create a small text file.
2. Use `shasum` to try the seven algorithms shown above on the file you created above.

Task 2: Checking Software Digests

In this task you will be learning how to verify the integrity of a downloaded file using hash functions. You will use the text-based "lynx" browser to download two files. While rudimentary, this browser is not vulnerable to most malicious attacks on browsers and may be suitable for retrieving files from web sites of questionable providence.

1. Type "lynx verydodgy.com" at the command prompt.
2. Use the arrow keys and "d" key to download and save the **`floppy57.fs`**
3. From that same web page select the **SHA256.sdx** file and download it. .

4. Return to the terminal and generate the SHA256 digest of the file you just downloaded to see if it matches the value contained in the SHA256.sdx file.

Note that item #1 of the worksheet asks a follow-up question.

---

[1] If you try to enter "man shasum", then you will get the man page for a Perl command instead of what you want, which is very confusing.

Task 3: Exploring the "Avalanche Effect"

The "avalanche effect" is a description for how a small change in the input file can cause a drastic effect on the output digest.

To understand the pseudo-random properties of cryptographic hash functions, do the following things:

1. Create a file named `iou.txt` with the following content: "Bob owes me 200 dollars".
2. Generate a SHA256 digest of the `iou.txt` file.
3. Open `iou.txt` with `and editor, e.g., leafpad .`
4. Change the '2' to a '3'.

   This will result in one bit being changed.

   An explanation for why a change from '2' to '3' results in one bit being changed.
   In the ASCII format, a '2' is the number 50, while a '3' is the number 51. The number 50 in binary is 00110010, while the number 51 in binary is 00110011; the only difference in these two numbers is the right-most binary digit.

5. Save your changes and exit the text editor.
6. Generate another SHA256 digest of the modified `iou.txt` file.

   Record in item #2 of the report your observations of the differences between the two digests of `iou.txt`.

7. Try a few more times to change iou.txt to see if you can get a new version of the file to match the digest of the original version.

   Record in item #3 of the report your experience with trying to get a different message to match the digest of the original data.

Note that item #4 asks a follow-up question.

Task 4: Exploring Second Pre-Image Resistance

In this task you will investigate the second pre-image resistant properties of the SHA256 cryptographic hash function (i.e., SHA2 with a 256-bit output). Because SHA256 is a relatively secure cryptographic hash function, you will not use its full 256-bit output. Instead, you will try to find a file that matches only a much smaller part of the digest.

1. First, generate a SHA256 digest for `declare.txt`.

   Record in item #5 of your report the last four hex digits of the displayed digest.

2. You will next be using a script named `collide1.sh`. The `collide1.sh` script was written to find random data that will hash to the same value as a given input file. Even better, it will let you specify how much of the digest you want to try to match (all or part).

   Execute the following command to find some random data that will hash such that the last hex digit of its digest will match the last hex digit of the digest for `declare.txt`.

   ```
   ./collide1.sh  declare.txt  1
   ```

   In item #6 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find a match on the last digit of the digest. Repeat the above command nine more times to fill out the table.

   Item #7 asks a follow-up question.

3. Now execute the following command to try to match the last two hex digits of the digest for `declare.txt`.

   ```
   ./collide1.sh  declare.txt  2
   ```

   In item #8 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find a match on the last two digits of the digest. Repeat the above command nine more times to fill out the table.

4. Now execute the following command to try to match the last three hex digits of the digest for `declare.txt`.

   ```
   ./collide1.sh  declare.txt  3
   ```

   In item #9 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find a match on the last three digits of the digest. Repeat the above command nine more times to fill out the table.

5. Transfer the information from the three completed tables to the spreadsheet named "Collide1.xlsx". A predefined graph will show your average results compared against the theoretical results.

   Item #10 asks a follow-up question about the graph shown in the completed spreadsheet.

Task 5: Exploring Collision Resistance

You will next be using a Python script called `collide2.py`. The `collide2.py` script tries to solve a different collision problem. It generates random data and hashes it, and then saves each digest in a table. It continues doing this until it has found two random strings of data whose digests match on the last byte (i.e., the last two hex digits).

1. Execute the above following command:

   ```
   ./collide2.py
   ```

   In item #11 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find two random messages whose digests match on the last byte. Repeat the above command nine more times to fill out the table.

   Item #12 asks a follow-up question.

Task 6 Exploring Message Authentication Codes

Within OpenSSL you will be using the hash-based message authentication code (HMAC) options. To get help on the command-line enter the following:

```
man dgst
```

You would enter the following to create a SHA1-based HMAC using OpenSSL:

```
openssl dgst -sha1 -hmac KEY FILENAME
```

replacing:

- *KEY* with any string of your choosing, **as long as it does not have spaces**. It appears that this key is first hashed in some fashion to create the actual key.
- *FILENAME* with the name of a file on your system

For example, to generate a SHA-1-based HMAC on a file named foo.txt (with a key of "mykey"), you would do the following:

```
openssl dgst -sha1 -hmac mykey foo.txt
```

There are other hash functions available other than "-sha1"; see "`man dgst`" for more information.

Do the following steps:

1. Use `openssl` as described above to create an HMAC for the file you created in Task 1.

2. Repeat the command with the same file but **use a different key** on the second try.

Record in item #13 of the report your observations about the HMAC outputs when different keys are used on the same file.

3. Suppose you intercepted a transmission that included: 1) `declare.txt` (one of the files in your home directory) and 2) its SHA1-based HMAC. The MAC tag you intercepted is:

   ```
   HMAC-SHA1(declare.txt)= 986eb8a92e561f550a911352c8b2cf5fd0465342
   ```

   Through some other means you find out that the key is **only** a single digit, i.e., one of the following: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

   Assume that you need to figure out the key (given the information learned above).

   Try all possible keys to determine the key that was used to generate the above HMAC value on the `declare.txt` file.

   Note that item #14 asks a follow-up question.

Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab macs-hash
```

If you modified the lab report or spreadsheet on a different system, you must copy those completed files into the directory paths displayed when you started the lab, and you must do that before typing "stoplab". When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.

Appendix – Some Unix Commands

`cd` Change the current directory.

```
cd destination
```

With no "destination" your current directory will be changed to your home directory. If you "destination" is "..", then your current directory will be changed to the parent of your current directory.

`cp` Copy a file.

```
cp source destination
```

This will copy the file with the "source" name to a copy with the "destination" name. The "destination" can also include the path to another directory.

`clear` Erase all the output on the current terminal and place the shell prompt at the top of the terminal.

`less` Display a page of a text file at a time in the terminal. (Also see `more`).

```
less file
```

To see another page press the space bar. To see one more line press the Enter key. To quit at any time press 'q' to quit.

`ls` List the contents and/or attributes of a directory or file

```
ls  location
ls  file
```

With no "location" or "file" it will display the contents of the current working directory.

`man` Manual

```
man command
```

Displays the manual page for the given "command". To see another page press the space bar. To see one more line press the Enter key. To quit before reaching the end of the file enter 'q'.

`more` Display a page of a text file at a time in the terminal. (Also see `less`).

```
more file
```

To see another page press the space bar. To see one more line press the Enter key. To quit at any time press 'q' to quit.

`mv` Move and/or Rename a file/directory

```
mv source destination
```

The "source" file will be moved and/or renamed to the given "destination."

`pwd` Display the present working directory

```
pwd
```

Lab 3 Report
Exploring MACs and Hash Functions

Your Name Here

Task 2: Checking Software Digests

1. In Task 2 you should have shown that the software you downloaded from the website was the same software posted on the website. What does it mean if they do not match? If an attacker could break into the website to replace the software with something malicious, what else would the attacker need to replace to get away with it?

   TBD

Task 3: Exploring the "Avalance Effect"

2. Describe the differences between the two digests of `iou.txt` when the difference between the two inputs is only one bit.

   TBD

3. Describe your experience to find another message that matches the original digest of `iou.txt`.

   TBD

4. Referring to your observations and experiences recorded in worksheet items #2 and #3, how does the avalanche effect make it difficult to find two messages that hash to the same value?

   TBD

Task 4: Exploring Second Pre-Image Resistance

5. The last four hex digits of the SHA256 digest of `declare.txt`: _____

6. The number of attempts to match on **the last hex digit** of the digest for `declare.txt`.

Attempt	# Tries
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

7. Referring to the data reported in item #6 above, why are the results so different amongst the ten attempts to find a match on the last digit of the digest? If a hex digit only has 16 possible outputs, why would it take more than 16 times to sometimes find a collision?

   TBD

8. The number of attempts to match on the last **two** hex digits of the digest for `declare.txt`.

Attempt	# Tries
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

9. The number of attempts to match on the last **three** hex digits of the digest for `declare.txt`.

Attempt	# Tries
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

10. Referring to the graph in the Excel spreadsheet, what kind of pattern do you see in the graph? What does this suggest if you tried to find a match against the entire hash for `declare.txt`?

    TBD

Task 5: Exploring Collison Resistance

11. The number of attempts to find two random messages whose digests match on the last byte (i.e., the last **two** hex digits).

Attempt	# Tries
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

12. Referring to the tables in items #8 and #11, when only two hex digits needed to match, explain why it required less effort to find a collision in #11 than it did in #8.

    TBD

Task 6: Exploring Message Authentication Codes

13. Describe your observations about the differences in the outputs when different keys are used to generate an HMAC for the same file.

    TBD

14. At the end of Task 6 you found a MAC key through a "brute force" effort. What could an adversary do if he could determine the MAC key that is used to protect the integrity of communications between two people?

    TBD

Other

15. Describe any experimentation that you performed.

    TBD

16. What did you learn from this exercise?

    TBD

17. How could this lab exercise be improved?

    TBD

[Page vierge]

PRIAMOS – SARL unipersonnelle au capital de 21000 € enregistrée au RCS de Toulouse
SIRET : 879959252 00021 | TVA : FR70879959252 | APE : 7022 Z | Organisme de formation n°76310985331
240 rue Jean Bart, 31290 Labège, France
+33 (0)6 46 91 92 13 | contact@priamos.fr | www.priamos.fr | www.linkedin.com/company/priamos/