Mercury

Generated by Doxygen 1.8.13

Contents

Chapter 1

Mercury: network fingerprinting and packet metadata capture

This package contains two programs for fingerprinting network traffic and capturing and analyzing packet metadata: **mercury**, a Linux application that leverages the modern Linux kernel's high-performance networking capabilities (AF_PACKET and TPACKETv3), which is described below, and ∗∗pmercury∗∗, a portable python application, which is described here.

Building and running mercury

In the root directory, run

```
./configure
make
```

to build the package and install the required pip3 modules (dpkt ujson numpy pyasn hpack pypcap). If you do not have **python3**, **cython**, and **pip3** installed, then you either need to install them (using apt, yum, or whatever your preferred package management tool is), or you need to run

```
./configure --disable-python
make
```

With the **–disable-python** flag, the configure script can build mercury in a way that omits the fingerprint analysis module (which is implemented using cython and python3). Without the analysis module, mercury can still perform fingerprint and metadata capture.

Compile-time options

There are compile-time options that can tune mercury for your hardware, or generate debugging output. Each of these options is set via a C/C++ preprocessor directive, which should be passed as an argument to "make". For instance, to turn on debugging, first run **make clean** to remove the previous build, then run **make "OPTFLAGS=-DDEBUG"**. This runs make, telling it to pass the string "-DDEBUG" to the C/C++ compiler. The available compile time options are:

- -DDEBUG, which turns on debugging, and

- -FBUFSIZE=16384, which sets the fwrite/fread buffer to 16,384 bytes. If multiple compile time options are used, then they must be passed to make together in the OPTFLAGS string, e.g. "OPTFLAGS=-DDEBUG -DFBUFSIZE=16384".

Running mercury

```
mercury: packet metadata capture and analysis
./src/mercury INPUT [OUTPUT] [OPTIONS]:
INPUT
   [-c or --capture] capture_interface   # capture packets from interface
   [-r or --read] read_file              # read packets from file
OUTPUT
   [-f or --fingerprint] json_file_name  # write fingerprints to JSON file
   [-w or --write] pcap_file_name        # write packets to PCAP/MCAP file
   no output option                      # write JSON packet summary to stdout
--capture OPTIONS
   [-b or --buffer] b                    # set RX_RING size to (b * PHYS_MEM)
   [-t or --threads] [num_threads | cpu] # set number of threads
   [-u or --user] u                      # set UID and GID to those of user u
--read OPTIONS
   [-m or --multiple] count              # loop over read_file count >= 1 times
GENERAL OPTIONS
   [-a or --analysis]                    # analyze fingerprints
   [-s or --select]                      # select only packets with metadata
   [-l or --limit] l                     # rotate JSON files after l records
   [-h or --help]                        # extended help, with examples
```

Details

[-c or –capture] c captures packets from interface c with Linux AF_PACKET using a separate ring buffer for each worker thread. **[-t or –thread] t** sets the number of worker threads to t, if t is a positive integer; if t is "cpu", then the number of threads will be set to the number of available processors. **[-b or –buffer] b** sets the total size of all ring buffers to (b $*$ PHYS_MEM) where b is a decimal number between 0.0 and 1.0 and PHYS_MEM is the available memory; USE b $<$ 0.1 EXCEPT WHEN THERE ARE GIGABYTES OF SPARE RAM to avoid OS failure due to memory starvation. When multiple threads are configured, the output is a *file set*: a directory into which each thread writes its own file; all packets in a flow are written to the same file.

[-f or –fingerprint] f writes a JSON record for each fingerprint observed, which incorporates the flow key and the time of observation, into the file or file set f. With **[-a or –analysis]**, fingerprints and destinations are analyzed and the results are included in the JSON output. The analysis output is documented in the pmercury README.

[-w or –write] w writes packets to the file or file set w, in PCAP format. With **[-s or –select]**, packets are filtered so that only ones with fingerprint metadata are written.

[r or –read] r reads packets from the file or file set r, in PCAP format. A single worker thread is used to process each input file; if r is a file set then the output will be a file set as well. With **[-m or –multiple] m**, the input file or file set is read and processed m times in sequence; this is useful for testing.

[-u or –user] u sets the UID and GID to those of user u; output file(s) are owned by this user. With **[-l or –limit] l**, each JSON output file has at most l records; output files are rotated, and filenames include a sequence number.

[-h or –help] writes this extended help message to stdout.

Examples

```
mercury -c eth0 -w foo.pcap           # capture from eth0, write to foo.pcap
mercury -c eth0 -w foo.pcap -t cpu    # as above, with one thread per CPU
mercury -c eth0 -w foo.mcap -t cpu -s # as above, selecting packet metadata
mercury -r foo.mcap -f foo.json       # read foo.mcap, write fingerprints
mercury -r foo.mcap -f foo.json -a    # as above, with fingerprint analysis
mercury -c eth0 -t cpu -f foo.json -a # capture and analyze fingerprints
```

Ethics

Mercury is intended for defensive network monitoring and security research and forensics. Researchers, administrators, penetration testers, and security operations teams can use these tools to protect networks, detect vulnerabilities, and benefit the broader community through improved awareness and defensive posture. As with any packet monitoring tool, Mercury could potentially be misused. **Do not run it on any network of which you are not the owner or the administrator**.

Credits

Mercury was developed by David McGrew, Brandon Enright, Blake Anderson, Shekhar Acharya, and Adam Weller, with input from Brian Long, Bill Hudson, and others. Pmercury was developed by Blake Anderson, with input from the others.

Acknowledgments

This software includes GeoLite2 data created by MaxMind, available from https://www.maxmind.com.

We make use of Mozilla's Public Suffix List which is subject to the terms of the Mozilla Public License, v. 2.0.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

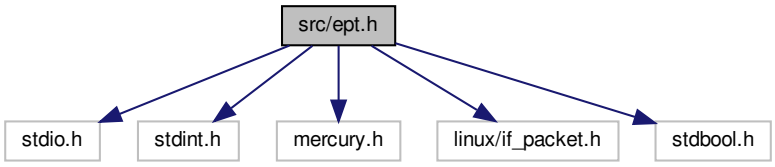
File Documentation

3.1 README.md File Reference

3.2 src/ept.h File Reference

```
#include <stdio.h>
#include <stdint.h>
#include "mercury.h"
```

Include dependency graph for ept.h:



Macros

- #define QUOTED_ASCII 0
- #define PARENT_NODE_INDICATOR 0x8000
- #define LENGTH_MASK 0x7fff

Functions

- void encode_uint16 (uint8_t *p, uint16_t x)
- uint16_t decode_uint16 (const void *x)
- enum status fprintf_binary_ept_as_paren_ept (FILE *f, const unsigned char *data, unsigned int len)
- void fprintf_binary_ept_as_tls_json (FILE *f, const unsigned char *data, unsigned int len)
- size_t sprintf_binary_ept_as_paren_ept (uint8_t *data, size_t length, unsigned char *outbuf, size_t outbuf↩ _len)
- size_t binary_ept_from_paren_ept (uint8_t *outbuf, const uint8_t *outbuf_end, const uint8_t *paren_ept, const uint8_t *paren_ept_end)
- enum status binary_ept_print_as_tls (uint8_t *data, size_t length)

3.2.1 Macro Definition Documentation

3.2.1.1 LENGTH_MASK

```
#define LENGTH_MASK 0x7fff
```

3.2.1.2 PARENT_NODE_INDICATOR

```
#define PARENT_NODE_INDICATOR 0x8000
```

3.2.1.3 QUOTED_ASCII

```
#define QUOTED_ASCII 0
```

Encoded Parse Tree (EPT)

An Encoded Parse Tree (EPT) is a general and flexible way to represent a fingerprint. It faithfully represents the implementation-static data extracted from packets, using a tree of byte strings that can be serialized in readable or binary form.

Selective packet parsing produces a parse tree whose leaves contain the selected fields of the packet. Each leaf (external node) of the parse tree contains a byte string, and each internal node holds an ordered list of nodes. The tree is serialized with a preorder traversal, so that the selected fields appear in the same order in the serialized data as they do on the wire. For a particular fingerprint format, the position of the node in the tree determines its semantics; for instance, the first leaf may be the version field, and so on.

Each protocol has its own fingerprint format, which specifies the fields that are extracted from the packets and how they are normalized. Because protocols evolve over time, fingerprint formats need to be able to evolve as well. We define formats for TLS, TCP, SSH, and DHCP.

The human-readable representation of an EPT uses bracket expressions. Each leaf is a hexadecimal string surrounded by parentheses, such as "(474554)", and each internal node is represented by a pair of balanced parenthesis that surround the nodes that it holds, such as "((0000)(000b00020100))". Bracket expressions are commonly used in natural language processing, and they can be easily parsed in a single pass. The binary representation of an EPT uses a simple type-length-value encoding that is defined below.

EPT is both flexible and reversible, meeting the following requirements:

- It can easily accommodate changes in the fingerprint format, and can readily be applied to new protocols,
- It can represent any selected fields from the fingerprint,
- It can support approximate fingerprint matching, so that slightly different fingerprints from slightly different applications can be matched,
- It avoids computationally expensive operations, such as cryptographic hashing, to facilitate use at high data rates,
- It provides both binary and human-readable forms. set QUOTED_ASCII to 1 for readable output like "(\"G↩ ET")"; set it to 0 for hex output like "(474554)"

3.2.2 Function Documentation

3.2.2.1 binary_ept_from_paren_ept()

```
size_t binary_ept_from_paren_ept (
            uint8_t * outbuf,
            const uint8_t * outbuf_end,
            const uint8_t * paren_ept,
            const uint8_t * paren_ept_end )
```

3.2.2.2 binary_ept_print_as_tls()

```
enum status binary_ept_print_as_tls (
            uint8_t * data,
            size_t length )
```

3.2.2.3 decode_uint16()

```
uint16_t decode_uint16 (
            const void * x )
```

3.2.2.4 encode_uint16()

```
void encode_uint16 (
            uint8_t * p,
            uint16_t x )
```

3.2.2.5 fprintf_binary_ept_as_paren_ept()

```
enum status fprintf_binary_ept_as_paren_ept (
            FILE * f,
            const unsigned char * data,
            unsigned int len )
```

3.2.2.6 fprintf_binary_ept_as_tls_json()

```
void fprintf_binary_ept_as_tls_json (
            FILE * f,
            const unsigned char * data,
            unsigned int len )
```

3.2.2.7 sprintf_binary_ept_as_paren_ept()

```
size_t sprintf_binary_ept_as_paren_ept (
            uint8_t * data,
            size_t length,
            unsigned char * outbuf,
            size_t outbuf_len )
```

3.3 src/libmerc.h File Reference

Functions

- size_t extract_fp_from_tls_client_hello (uint8_t *data, size_t data_len, uint8_t *outbuf, size_t outbuf_len)
  extracts a TLS client fingerprint from a packet
- enum status proto_ident_config (const char *config_string)
- enum status static_data_config (const char *config_string)

3.3.1 Function Documentation

3.3.1.1 extract_fp_from_tls_client_hello()

```
size_t extract_fp_from_tls_client_hello (
            uint8_t * data,
            size_t data_len,
            uint8_t * outbuf,
            size_t outbuf_len )
```

extracts a TLS client fingerprint from a packet

Extracts a TLS clientHello fingerprint from the TCP data field (which starts at *data* and contains *data_len* bytes) and writes it into the output buffer (which starts at *outbuf* and contains *outbuf_len* bytes) in bracket notation (human readable) form, if there is enough room for it.

Parameters

in	*data*	the start of the TCP data field
in	*data_len*	the number of bytes in the TCP data field
out	*outbuf*	the output buffer
in	*outbuf_len*	the number of bytes in the output buffer

3.3.1.2 proto_ident_config()

```
enum status proto_ident_config (
             const char * config_string )
```

3.3.1.3 static_data_config()

```
enum status static_data_config (
             const char * config_string )
```

Index