A Package for Capturing and Analyzing Network Data Features

TLS Fingerprinting Addendum

Joy version 4.0
February 2019

Preface

Joy is a BSD-licensed open source software package for collecting and analyzing network data, with a focus on network data feature exploration. This document shows how it can be used, installed, built, and modified. We hope that you find it useful, and that you enjoy using it. Please do understand that it is open source software, with the following licensing terms:

Contents

Chapter 1

Introduction

1.1 Overview

TLS Fingerprinting is a technique that associates parameters extracted from a TLS ClientHello with a database of known fingerprints to provide visibility into the application and/or TLS library that created the session. Applications of TLS fingerprinting include malware detection [3], minor-version operating system identification [2], and application identification. TLS fingerprinting has been studied since at least 2012 [7], and several open source databases have been released [1, 5, 6].

We extend the previous work on two fronts. First, as explained below, we developed a continuous pipeline that correlates endpoint and network data to ensure that our fingerprint database stays up-to-date. Second, we have focused on extracting all of the discriminating features in the ClientHello and providing more context when a TLS fingerprint matches a entry in our database. We use the automatic endpoint/network data correlation to provide lists of processes, along with their sha256 and prevalence, that we have seen associated with a TLS fingerprint. Additional details are provided in the schema definition (Chapter 2).

Our open source contributions include a fingerprint database that is updated weekly, a Joy feature to generate fingerprint strings for TLS flows, and a set of python programs to identify TLS fingerprints in a packet capture or off a live network interface, generate new TLS fingerprints that can be contributed to the open source community, and a web-based user interface to visualize the results of TLS fingerprinting.

1.2 TLS Fingerprinting

Given a typical TLS session as shown in Figure 1.1, TLS fingerprinting extracts features from the first application data packet sent by the client, i.e., the `ClientHello`. The `ClientHello` message provides the server with a list of cipher suites and extensions that the client supports. The cipher suite list is ordered by preference of the client, and each cipher suite defines a set of cryptographic algorithms needed for TLS to operate. The TLS-defined cryptographic algorithms have real-world trade-offs, and this is reflected in different applications choosing to omit, include, or prefer cipher suites more aligned with their goals. The set of extensions provides additional information to the server that facilitates extended functionality, e.g., the Application Layer Protocol Negotiation extension indicates the list of application layer protocols that the client supports, e.g., `h2` and `http/1.1`. Some data carried in the extensions are useful for identifying the client application, while other extension data is session specific, e.g., the `SessionTicket` data. The string representation and unique identifier for our TLS fingerprints are in the following format:

```
(version)(cipher suites)((extensions)...)
```

where:

- `(version)` is the hex representation of the advertised version, e.g., `(0303)` for TLS 1.2.

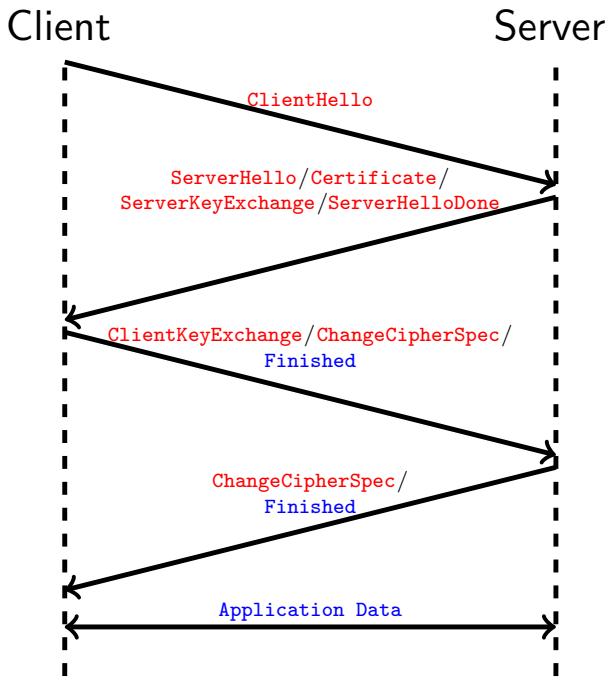

Figure 1.1: Graphical representation of a simplified TLS handshake and application data protocols. TLS fingerprinting uses features taken from the unencrypted `ClientHello` message.

- `(cipher suites)` is a hex list of the cipher suite offer vector, e.g., `(c02cc02bc030...0013)`.
- `((extensions)...)` is a nested list of the hex-encoded extensions, where some extensions will have their data removed. Each individual extension with data looks like `(000b00020100)` where the first two bytes, `000b`, represent the extension hex code; the second two bytes, `0002`, represent the hex-encoded length of the data; and the last set of bytes, `0100`, represent the hex-encoded extension data. For extensions where we discard the data, we just report the hex-encoded extension type code, e.g., `(0000)`.

We include static extension data for `supported_groups`, `ec_point_formats`, `status_request`, `signature_algorithms`, `application_layer_protocol_negotiation`, `supported_versions`, and `psk_key_exchange_modes`.

`GREASE` [4] is a method to ensure TLS servers are properly handling unknown TLS cipher suites, extensions, and extension data. It is defined as a set of 16 hex codes, {`0a0a,...,fafa`}, that can be advertised in the ClientHello, but should never be chosen by the server. For our fingerprinting database, we normalize `GREASE` data to `0a0a`, but do not change the position of the data. Therefore, we avoid creating duplicate fingerprints based on the random `GREASE` hex code selection, but can still leverage the information about how the client is using `GREASE`.

In our current studies, TLS 1.3 has improved the efficacy of TLS fingerprinting due to the added data features in the ClientHello. While, there are currently only 5 cipher suites defined for TLS 1.3, most TLS clients released in the foreseeable future will be backwards compatible with TLS 1.2, and will therefore offer many "legacy" cipher suites. In addition to the 5 cipher suites, there are several new extensions. `supported_versions` is an interesting TLS 1.3 extensions that allows us to differentiate between clients that supported earlier draft implementations of TLS 1.3.

1.3 Data Collection

The visibility gained by TLS fingerprinting is highly dependent on the underlying fingerprint database, and until now, generating this database was a manual process that was slow to update and was not

reflective of real-world TLS usage. Building on work we first publically released in July 2016 [3], we created a continuous process that fuses network telemetry with endpoint telemetry to build fingerprint databases automatically. This allows us to provide detailed information for a given fingerprint that is representative of how that fingerprint is used in the real-world, e.g., sha256's and lists of processes associated with a fingerprint sorted by the prevalence those applications were observed on the network's we monitor.

Chapter 2

Fingerprint Database Schema

2.1 Overview

The Fingerprint Schema is composed of 4 main sections. The first set of metadata fields provides some general information about the fingerprint, including unique keys that can be used to index the fingerprint entry in a dictionary. "tls_features" contains a human-readable description of the cipher suites, extensions, and extension data that was used to generate the TLS fingerprint. Finally, "process/os_info" contain information regarding the three most prevalent processes and operating systems we have observed using a given fingerprint.

Figure 2.1: Schema Overview

```
.
{
  "str_repr": "...",                // Fingerprint metadata objects
  "md5_repr": "...",
  "source": ["..."],
  "max_implementation_date": "...",
  "min_implementation_date": "...",
  "tls_features": {...},            // Human-readable description of TLS features
  "process_info": [...],            // List of associated process information objects
  "os_info": [...]                  // List of associated OS information objects
}
```

2.2 Metadata

The metadata fields are meant to provide a high-level summary of the fingerprint. "str_repr" is a string representation of the hex-values for the TLS features. It is reversible in the sense that there is a one-to-one correspondence with this representation and what you would observe on the wire, with a single caveat: all GREASE values are encoded as 0a0a. "md5_repr" is the MD5 hash of the string representation, and source provides information about where a fingerprint was collected. We generate "max/min_implementation_date" by first correlating every TLS parameter with the date of the RFC that first defined that parameter. From this list of dates, we report both the oldest and newest.

Figure 2.2: Metadata Fields

```
.
{
  "str_repr": "(0301)(003900380033003200350002f00ff)((0000)(0023))",
  "md5_repr": "89f96e9f19fd8754d9965ede7937a7b3",
  "source": ["Cisco"],
  "max_implementation_date": "2010-02",
  "min_implementation_date": "2002-06",
  ...
}
```

2.3 TLS Features

"tls_features" is simply a human-readable description of the TLS features we use to construct the TLS fingerprint. The names should correspond to the names defined in the appropriate RFCs, but in some instances the parameters are unrecognized by a standards body and/or unknown to us. In these cases, we report an unknown identifier with the hex value.

Figure 2.3: TLS Fields

```
.
{
  ...
  "tls_features": {
    "version": "TLS 1.0",
    "cipher_suites": [
      "TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
      "TLS_DHE_DSS_WITH_AES_256_CBC_SHA",
      "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
      "TLS_DHE_DSS_WITH_AES_128_CBC_SHA",
      "TLS_RSA_WITH_AES_256_CBC_SHA",
      "TLS_RSA_WITH_AES_128_CBC_SHA",
      "TLS_EMPTY_RENEGOTIATION_INFO_SCSV"
    ],
    "extensions": [
      {"server_name": ""},
      {"session_ticket": ""}
    ]
  },
  ...
}
```

2.4 Process Information

We report the three most prevalent processes that we have seen using a given TLS fingerprint. For each process, we provide the name, sha256, and the prevalence. In Figure 2.4, a prevalence of 0.66 means that two thirds of the time, this specific fingerprint is observed coming from the process with the given sha256. Optionally, we have assigned a number of processes to application categories, which we report if available.

Figure 2.4: Process Information Fields

```
.
{
  ...
  "process_info": [
    {
      "process": "Python",
      "application_category": "programming",
      "prevalence": 0.66,
      "sha256": "A45244F6DCF841A6C165438BB16D074487A3BB1E83D705CBFB24690C3E09DCAE"
    },
    ...
  ],
  ...
}
```

2.5 Operating System Information

Similar to the process information object, we report the three most prevalent operating systems that we have seen using a given TLS fingerprint. We report the general operating system, the OS version and edition, and the prevalence we have seen the OS being used to generate the given TLS fingerprint.

Figure 2.5: OS Information Fields

```
.
{
  ...
  "os_info": [
    {
      "os": "Mac OS X",
      "os_version": "10.11.6",
      "os_edition": "El Capitan",
      "prevalence": 0.99
    },
    ...
  ]
}
```

Chapter 3

The `fingerprinter` Tool

`fingerprinter.py` is a stand-alone python program that will read either a pcap file or a live network interface and assign a TLS fingerprint to every ClientHello it observes. If the TLS fingerprint has an exact match in the supplied database, then the process and OS information of the exact match will be reported. If the fingerprint is not in the database, but has a "close" approximate match, the process and OS information of the approximate match will be reported. The approximate match will then be added to the in-memory version of the database to facilitate more efficient future lookups.

3.1 Dependencies

```
pypcap >= 1.1.6
dpkt   >= 1.9.1
numpy  >= 1.14.2
```

```
pip install pypcap
pip install dpkt
pip install numpy
```

3.2 Examples

```
./fingerprinter.py test.pcap

{"source_addr": "10.0.2.15", "dest_addr": "216.58.217.165", "source_port":
34496, "dest_port": 443, "timestamp": "2018-06-04 14:52:18.430744",
"fingerprint": {...}}
...
```

3.3 Command Line Options

3.3.1 -i INPUT, –input=INPUT

> Syntax:
>
> `-i INPUT, --input=INPUT`

The command `-i INPUT` specifies the pcap file or interface.

3.3.2 -f FP_DB, –fp_db=FP_DB

> Syntax:
>
> `-f FP_DB, --fp_db=FP_DB`

The command `-f FP_DB` specifies the location of fingerprint database (e.g., resources/fingerprint_db.json.gz).

3.3.3 -p PORT, –port=PORT

> Syntax:
>
> `-p PORT, --port=PORT`

The command `-p PORT` causes `fingerprinter` to only process traffic destined to port `PORT`.

3.3.4 -o OUTPUT, –output=OUTPUT

> Syntax:
>
> `-o OUTPUT, --output=OUTPUT`

The command `-o OUTPUT` specifies the output file (the default is stdout).

3.3.5 -l LOOKUP, –lookup=LOOKUP

> Syntax:
>
> `-l LOOKUP, --lookup=LOOKUP`

The command `-l LOOKUP` reports the database entry for a given fingerprint string, `str_repr`.

Chapter 4

The gen_tls_fingerprint Tool

gen_tls_fingerprint.py is a tool to facilitate the small-scale generation of TLS fingerprint databases that could be easily merged with the current open source fingerprint database hosted on the Joy github site. This program uses the output of the Joy network monitoring tool with two enabled features: fpx and exe. fpx reports the fingerprint string representation for each observed TLS flow containing a ClientHello, and exe hooks into the host operating system to report information about the process that generated the TLS flow. The resulting database can optionally include contributor information that would be integrated into the main open source fingerprint database.

4.1 Dependencies

```
numpy >= 1.14.2
```

```
pip install numpy
```

4.2 Examples

```
./gen_tls_fingerprint.py fp_test.json.gz

{"str_repr": "...", "tls_features": {...}, "process_info": [...]}
{"str_repr": "...", "tls_features": {...}, "process_info": [...]}
{"str_repr": "...", "tls_features": {...}, "process_info": [...]}
...
```

4.3 Command Line Options

4.3.1 -i INPUT, –input=INPUT

> Syntax:
>
> `-i INPUT, --input=INPUT`

The command `-i INPUT` specifies the Joy JSON output, with fpx=1 and exe=1.

4.3.2 -o OUTPUT, –output=OUTPUT

> Syntax:
>
> `-o OUTPUT, --output=OUTPUT`

The command `-o OUTPUT` specifies the output file for the generated fingerprint database.

4.3.3 -c CONTRIB_INFO, –contrib_info=CONTRIB_INFO

> Syntax:
>
> `-c CONTRIB_INFO, --contrib_info=CONTRIB_INFO`

The command `-c CONTRIB_INFO` specifies brief contributor information to be included in the fingerprint source.

Chapter 5

The `fingerprint_ui` Tool

`fingerprint_ui.py` provides a simple, bottle-based user interface to inspect packet capture files. It will read the uploaded pcap file, extract the TLS fingerprint for all relevant connections, identify matches in the fingerprint database, and present the information in tabular form. A modal provides detailed information about the matching fingerprint and TLS parameters.

5.1 Dependencies

```
bottle >= 0.12.13
pypcap >= 1.1.6
dpkt   >= 1.9.1
numpy  >= 1.14.2
```

```
pip install bottle
pip install pypcap
pip install dpkt
pip install numpy
```

5.2 Examples

```
./fingerprint_ui.py
```

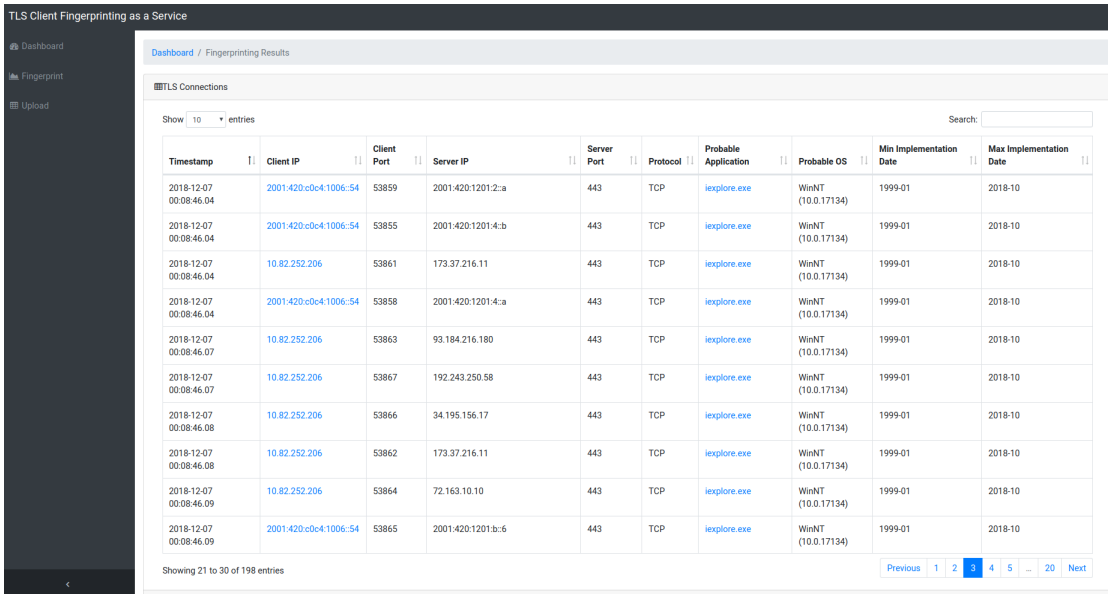

Figure 5.1: Fingerprint Results from Analyzing a pcap File.

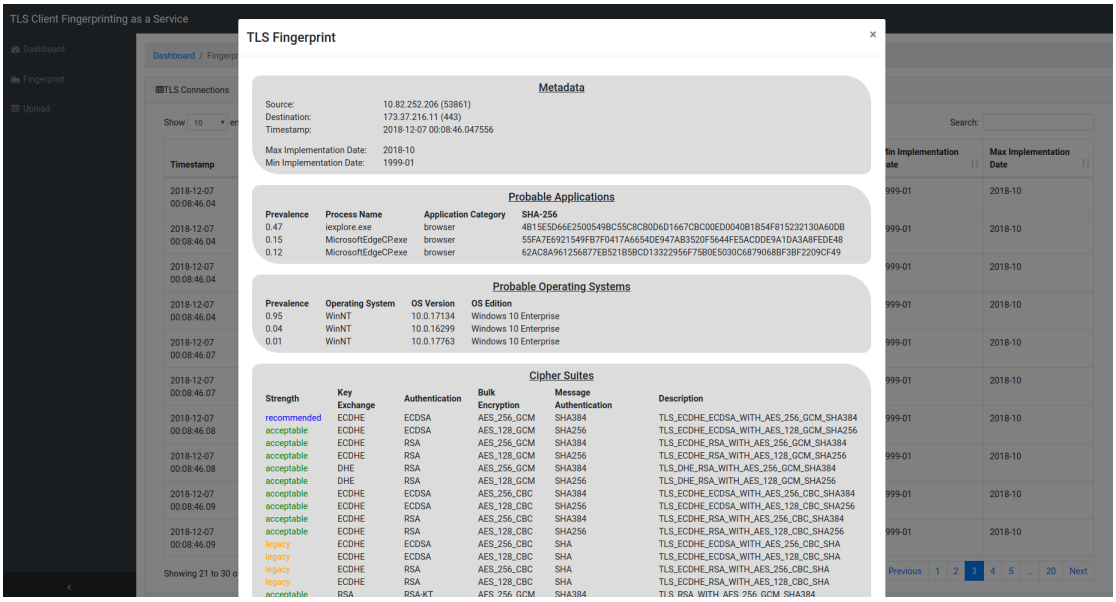

Figure 5.2: Fingerprint Inspection

Bibliography

[1] John B. Althouse, Jeff Atkinson, and Josh Atkins. JA3. https://github.com/salesforce/ja3.

[2] Blake Anderson and David McGrew. OS Fingerprinting: New Techniques and a Study of Information Gain and Obfuscation. In *IEEE Conference on Communications and Network Security (CNS)*, 2017.

[3] Blake Anderson, Subharthi Paul, and David McGrew. Deciphering Malware's Use of TLS (without Decryption). *Journal of Computer Virology and Hacking Techniques*, pages 1–17, 2017.

[4] David Benjamin. Applying GREASE to TLS Extensibility. Internet-Draft (Informational), 2017. https://www.ietf.org/archive/id/draft-ietf-tls-grease-00.txt.

[5] Lee Brotherston. FingerprinTLS. https://github.com/LeeBrotherston/tls-fingerprinting.

[6] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of Age: A Longitudinal Study of TLS Deployment. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 415–428, 2018.

[7] Marek Majkowski. SSL Fingerprinting for p0f. https://idea.popcount.org/2012-06-17-ssl-fingerprinting-for-p0f/.