DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC

Gaganjeet Reen and Christian Rossow
CISPA Helmholtz Center for Information Security
• Transport protocol that has the potential to replace TCP in the near future.

• Encrypted-by-default. Conceptually similar to a combination of TCP, TLS, and HTTP/2 implemented on UDP.
An in-line proxy intercepts QUIC communication and forwards decrypted QUIC communication. The Attacker (e.g., insider) wants to bypass the DPI, and potential strategies to achieve this are discussed.
We use 5 popular open source implementations to model a DPI system.

To model an inline proxy that forwards the packets to a DPI system for further inspection, we feed same logical sequence of packets to the DPI system and the Implementation Under Test (IUT).

Uses fuzzing to generate sequences
**Challenges**

- **Challenge**: Implementations of protocols like QUIC expect packets with very specific structure and field values in the absence of which, the packets are discarded right away. A fuzzer will only be able to detect meaningful vulnerabilities if it can inject unexpected inputs deep into the state space of a protocol.
  
  **Solution**: We design a “smart” fuzzer which has structure-aware “generators” and “mutators”.

- **Challenge**: QUIC is an encrypted by default protocol. Fuzzing encrypted packets will almost certainly render a packet structurally invalid.
  
  **Solution**: We fuzz packets before they are encrypted.

- **Challenge**: Cannot replay previously captured traffic because completing a handshake is essential before any data can be exchanged.
  
  **Solution**: Our fuzzer is capable of (i) completing handshakes and (ii) interacting actively with the implementations.
Mutations

Packet
- Payload
  - Repeat
  - Alter
  - Add
  - Drop
- Frame
- Fields

Sequence
- Shuffle
- Duplicate
- Drop
Packet Generators

- Randomised
- Controlled
  - Basic Stream Reassembly
  - Flow Control Aware Stream Reassembly
  - Overlapping Offset
DPIFuzz Architecture

DPIFuzz

Deterministic RNG

Seed Generator

Implementations to test

Generator List

Iterations

Connection Handler

DPIFuzz

QUICLY
QUANT
QUICHE
NEQO
MVFST

QUIC Implementations

Response Analyzer

Trace Files

Differential Analysis

Result File
The experiments were run on a machine with a Quad-Core Intel i5 processor with a 16GB RAM running Ubuntu 18.04.

To remove the effects of network latency and unintentional packet reordering, we run our echo servers as well as DPIFuzz locally.

In total, we use our framework to create 600 unique sequences, i.e., we run our fuzzer against each IUT, with each of the 3 specified generators, with 200 different seed values.

<table>
<thead>
<tr>
<th>Transport Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial_max_data</td>
<td>1048576</td>
</tr>
<tr>
<td>initial_max_stream_data_bidi_local</td>
<td>66560</td>
</tr>
<tr>
<td>initial_max_stream_data_bidi_remote</td>
<td>66560</td>
</tr>
<tr>
<td>initial_max_stream_data_uni</td>
<td>66560</td>
</tr>
<tr>
<td>initial_max_streams bidi</td>
<td>2048</td>
</tr>
<tr>
<td>initial_max_streams uni</td>
<td>2048</td>
</tr>
<tr>
<td>max_idle_timeout</td>
<td>60000</td>
</tr>
<tr>
<td>max_packet_size</td>
<td>1500</td>
</tr>
<tr>
<td>ack_delay_exponent</td>
<td>3</td>
</tr>
</tbody>
</table>
Reassembly Differences

- Exploiting packets with **duplicate packet numbers**

<table>
<thead>
<tr>
<th>Seed Value</th>
<th>Generator</th>
<th>QUICHE Reassembled Data</th>
<th>MVFST Reassembled Data</th>
<th>QUICLY Reassembled Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4373445819122772715</td>
<td>Basic Stream Reassembly</td>
<td>2Z?b”?$@N5?$#48SZyBp????:yGJ-+$0P7cdWYaBMetcem=+@-Wf-Sja1xZwhYKFFA26AN&amp;YI_</td>
<td>yGJ+$0P7cdWYaBMetcem=+@-Wf-Sja1xZwhYKFFA26AN&amp;YI_</td>
<td>yGJ+$0P7cdWYaBMetcem=+@-Wf-Sja1xZwhYKFFA26AN&amp;YI_</td>
</tr>
</tbody>
</table>

- The QUICHE server reassembles data from 2 streams while the QUICLY and MVFST servers only reassemble data from 1 stream. This happens because the sequence includes a packet with a duplicate packet number.
• Exploiting **Stream Offset Overlaps**

<table>
<thead>
<tr>
<th>Packet No.</th>
<th>Stream Frame Payload</th>
<th>Stream Offset</th>
<th>Stream Finbit</th>
<th>QUICHE Reassembled Data (Destination Server)</th>
<th>QUICLY Reassembled Data (DPI system)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCKED</td>
<td>2</td>
<td>True</td>
<td>_OCKED</td>
<td>_OCKED</td>
</tr>
<tr>
<td>2</td>
<td>BLIN</td>
<td>0</td>
<td>False</td>
<td>BLOCKED</td>
<td>BLINKED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packet No.</th>
<th>Stream Frame Payload</th>
<th>Stream Offset</th>
<th>Stream Finbit</th>
<th>MVFST Reassembled Data (Destination Server)</th>
<th>QUICLY Reassembled Data (DPI system)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCKED</td>
<td>2</td>
<td>True</td>
<td>_OCKED</td>
<td>_OCKED</td>
</tr>
<tr>
<td>2</td>
<td>INKED</td>
<td>2</td>
<td>False</td>
<td>_INKED</td>
<td>_INKED</td>
</tr>
<tr>
<td>3</td>
<td>BL</td>
<td>0</td>
<td>False</td>
<td>BLOCKED</td>
<td>BLINKED</td>
</tr>
</tbody>
</table>

...
Bugs and Vulnerabilities

<table>
<thead>
<tr>
<th>Seed Value</th>
<th>Generator</th>
<th>Implementation</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5224880393376231849</td>
<td>Basic Stream Reassembly</td>
<td>MVFST</td>
<td>Null Pointer Dereference</td>
</tr>
<tr>
<td>6867396659762739268</td>
<td>Flow-Control-Aware Stream Reassembly</td>
<td>QUANT</td>
<td>Heap use after free</td>
</tr>
<tr>
<td>3544824671711368728</td>
<td>Flow-Control-Aware Stream Reassembly</td>
<td>QUICLY</td>
<td>Null Pointer Dereference</td>
</tr>
<tr>
<td>8969571667189322506</td>
<td>Basic Stream Reassembly</td>
<td>NEQO</td>
<td>Assertion Failed</td>
</tr>
</tbody>
</table>

- **MVFST**: Unable to handle a stream frame in which the Offset field has a non-zero value but value of the OFF bit is set to 0 (which indicates that the Offset field is absent), the FIN bit is set to true and the Payload field empty.
- **QUANT**: Implementation tries to access the state of a stream, i.e., check whether a stream is closed or not after the memory allocated to the stream has already been freed.
- **QUICLY**: Unable to handle stream IDs not permitted by the server.
- **NEQO**: Attempts to close an already closed connection.
Thank You!

reen.gagan@gmail.com

https://github.com/piano-man/DPIFuzz