Analysis of QUIC Session Establishment and its Implementations

Eva Gagliardi\textsuperscript{1,2} Olivier Levillain\textsuperscript{1}

\textsuperscript{1}Télécom SudParis
\textsuperscript{2}French Ministry of the Armies

Séminaire SoSySec
May 29th 2020
Introduction

QUIC in a Nutshell

QUIC Packet Protection

A Look at QUIC Draft 23 Implementations

Conclusion and Perspectives
Introduction

QUIC in a Nutshell

QUIC Packet Protection

A Look at QUIC Draft 23 Implementations

Conclusion and Perspectives
Olivier Levillain

- M2 internship on the FORK-256 hash function (2006)
- Member of the systems security lab at ANSSI (2007-2012)
- Head of the network security lab at ANSSI (2012-2015)
- Head of the training center at ANSSI (2015-2018)
- Associate professor at Télécom SudParis (2018-)
Olivier Levillain

- M2 internship on the FORK-256 hash function (2006)
- member of the systems security lab at ANSSI (2007-2012)
- head of the network security lab at ANSSI (2012-2015)
- head of the training center at ANSSI (2015-2018)
- associate professor at Télécom SudParis (2018-)

Research

- low-level security mechanisms in x86 CPUs (ACPI, SMM)
- PhD on SSL/TLS
- studies on the languages
- work on parsers and on network protocol implementations
Documents and tools

https://paperstreet.picty.org

- my PhD manuscript (if you are into TLS)
- articles and slides for most of my contributions and seminars
Documents and tools

https://paperstreet.picty.org
- my PhD manuscript (if you are into TLS)
- articles and slides for most of my contributions and seminars

Active software projects
- Parsifal, a parser generator written in OCaml
  - https://github.com/picty/concerto
- Concerto, a tool to analyse TLS campaigns and certificate chains
  - https://github.com/picty/parsifal
- Wombat, one more Bleichenbacher toolkit
  - https://gitlab.com/pictyeye/wombat
The GASP project


▶ description of protocol messages using simple languages
▶ network scans at large to better understand real world ecosystems
▶ description of protocol state machines using simple languages
▶ security evaluation of concrete implementation using different techniques (message-level fuzzing, state machine inference)
The GASP project


▶ description of protocol messages using simple languages
▶ network scans at large to better understand real world ecosystems
▶ description of protocol state machines using simple languages
▶ security evaluation of concrete implementation using different techniques (message-level fuzzing, state machine inference)

Work in progress

▶ a platform to test and compare parser generators
▶ experimentations to fuzz existing state machines with L*  
  ▶ reproduction of existing results on TLS
  ▶ extension to the discovery of Bleichenbacher oracles
  ▶ performance improvement
▶ application to DNS, TLS, QUIC, SSH
The GASP project


- description of protocol messages using simple languages
- network scans at large to better understand real world ecosystems
- description of protocol state machines using simple languages
- security evaluation of concrete implementation using different techniques (message-level fuzzing, state machine inference)

Work in progress

- a platform to test and compare parser generators
- experimentations to fuzz existing state machines with L*
  - reproduction of existing results on TLS
  - extension to the discovery of Bleichenbacher oracles
  - performance improvement
- application to DNS, TLS, QUIC, SSH
Warnings about this presentation

Most of the material presented here comes from the work from Eva Gagliardi (2019 internship) and was presented at WISTP last December.
Warnings about this presentation

Most of the material presented here comes from the work from Eva Gagliardi (2019 internship) and was presented at WISTP last December.

The experiments were made against draft-23 implementations and may not accurately reflect on the current state of the ecosystem (current version is draft-28, mostly with minor changes regarding the session establishment).
Introduction

QUIC in a Nutshell

QUIC Packet Protection

A Look at QUIC Draft 23 Implementations

Conclusion and Perspectives
gQUIC and QUIC in a nutshell

▶ 2012: Google proposes a new protocol, QUIC
  ▶ multiplexed HTTP in a secure channel over UDP
gQUIC and QUIC in a nutshell

- 2012: Google proposes a new protocol, QUIC
  - multiplexed HTTP in a secure channel over UDP
- 2014: First drafts about TLS 1.3, borrowing some ideas
gQUIC and QUIC in a nutshell

- 2012: Google proposes a new protocol, QUIC
  - multiplexed HTTP in a secure channel over UDP
- 2014: First drafts about TLS 1.3, borrowing some ideas
- 2016: QUIC is proposed as an IETF item
  - the original protocol is renamed gQUIC
  - a new IETF WG is formed (quic)
  - a more modular design is proposed, with the soon-to-be TLS 1.3 as the secure transport
gQUIC and QUIC in a nutshell

- 2012: Google proposes a new protocol, QUIC
  - multiplexed HTTP in a secure channel over UDP
- 2014: First drafts about TLS 1.3, borrowing some ideas
- 2016: QUIC is proposed as an IETF item
  - the original protocol is renamed gQUIC
  - a new IETF WG is formed (quic)
  - a more modular design is proposed, with the *soon-to-be* TLS 1.3 as the secure transport
- 2018: TLS 1.3 publication (RFC8446)
gQUIC and QUIC in a nutshell

- 2012: Google proposes a new protocol, QUIC
  - multiplexed HTTP in a secure channel over UDP
- 2014: First drafts about TLS 1.3, borrowing some ideas
- 2016: QUIC is proposed as an IETF item
  - the original protocol is renamed gQUIC
  - a new IETF WG is formed (quic)
  - a more modular design is proposed, with the soon-to-be TLS 1.3 as the secure transport
- 2018: TLS 1.3 publication (RFC8446)
- 2019-2020: ongoing work on QUIC drafts (leading to -draft28 versions)
gQUIC and QUIC in a nutshell

- 2012: Google proposes a new protocol, QUIC
  - multiplexed HTTP in a secure channel over UDP
- 2014: First drafts about TLS 1.3, borrowing some ideas
- 2016: QUIC is proposed as an IETF item
  - the original protocol is renamed gQUIC
  - a new IETF WG is formed (quic)
  - a more modular design is proposed, with the soon-to-be TLS 1.3 as the secure transport
- 2018: TLS 1.3 publication (RFC8446)
- 2019-2020: ongoing work on QUIC drafts (leading to -draft28 versions)

Warning: this presentation is about IETF QUIC only
A Typical QUIC Connection

Client

Server

Uses UDP Packets

QUIC Initial (ClientHello)

QUIC Initial (ServerHello)

QUIC Handshake (EncryptedExtensions + Certificate + CertVerify + Finished)

QUIC Handshake (Finished)

QUIC 1 RTT (Application data)

Uses UDP Packets

QUIC in a Nutshell
A Typical QUIC Connection

Client

QUIC Initial
(ClientHello)

Server

Uses UDP Packets

Initial Protection
A Typical QUIC Connection

QUIC in a Nutshell
A Typical QUIC Connection

QUIC Initial (ClientHello)

QUIC Initial (ServerHello)

QUIC Handshake (EncryptedExtensions + Certificate + CertVerify + Finished)

Initial Protection

Handshake Secrets

Uses UDP Packets
A Typical QUIC Connection

- **QUIC Initial (ClientHello)**
- **QUIC Initial (ServerHello)**
- **QUIC Handshake (EncryptedExtensions + Certificate + CertVerify + Finished)**
- **QUIC Handshake (Finished)**

Uses UDP Packets

Initial Protection

Handshake Secrets
A Typical QUIC Connection

- **QUIC Initial** (ClientHello)
- **QUIC Initial** (ServerHello)
- **QUIC Handshake** (EncryptedExtensions + Certificate + CertVerify + Finished)
- **QUIC Handshake** (Finished)
- **QUIC 1 RTT** (Application data)

Uses UDP Packets

AppData after 1 RTT

Initial Protection

Handshake Secrets

Traffic Secrets
1RTT?

The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged.
1RTT?

The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged

- TLS (≤ 1.2) typically offers 3 RTT (TCP + 2)...
- 2 RTT (TCP + 1) with session resumption
- TLS 1.3 typically offers 2 RTT (TCP + 1)...
- 1 RTT (TLS 1.3 0 RTT mode) with session resumption, under conditions
- QUIC, thanks to UDP, is really 1 RTT in common cases...
- or even 0 RTT under conditions

However, do not forget that TCP is not slow on purpose, and that connection-oriented communications have benefits
1RTT?

The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged

- TLS ($\leq 1.2$) typically offers 3 RTT (TCP + 2)...
  - 2 RTT (TCP + 1) with session resumption

QUIC, thanks to UDP, is really 1 RTT in common cases... or even 0 RTT under conditions...
1RTT?

The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged

- TLS ($\leq 1.2$) typically offers 3 RTT (TCP + 2)...
  - 2 RTT (TCP + 1) with session resumption
- TLS 1.3 typically offers 2 RTT (TCP + 1)...
- QUIC, thanks to UDP, is really 1 RTT in common cases...
  - or even 0 RTT under conditions

However, do not forget that TCP is not slow on purpose, and that connection-oriented communications have benefits
The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged.

- **TLS (≤ 1.2)** typically offers 3 RTT (TCP + 2)...
  - 2 RTT (TCP + 1) with session resumption
- **TLS 1.3** typically offers 2 RTT (TCP + 1)...
  - 1 RTT (TLS 1.3 0 RTT mode) with session resumption, under conditions

**1RTT?**
The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged.

- **TLS (≤ 1.2)** typically offers 3 RTT (TCP + 2)...
  - 2 RTT (TCP + 1) with session resumption
- **TLS 1.3** typically offers 2 RTT (TCP + 1)...
  - 1 RTT (TLS 1.3 0 RTT mode) with session resumption, under conditions
- **QUIC**, thanks to UDP, is really 1 RTT in common cases...
1RTT?

The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged:

- **TLS (≤ 1.2)** typically offers 3 RTT (TCP + 2)...
  - 2 RTT (TCP + 1) with session resumption
- **TLS 1.3** typically offers 2 RTT (TCP + 1)...
  - 1 RTT (TLS 1.3 0 RTT mode) with session resumption, under conditions
- **QUIC**, thanks to UDP, is really 1 RTT in common cases...
  - or even 0 RTT under conditions
The efficiency of the session establishment is usually measured in the number of Round-Trip Times (RTTs) required before the first application data can be exchanged.

- TLS (≤ 1.2) typically offers 3 RTT (TCP + 2)...
  - 2 RTT (TCP + 1) with session resumption
- TLS 1.3 typically offers 2 RTT (TCP + 1)...
  - 1 RTT (TLS 1.3 0 RTT mode) with session resumption, under conditions
- QUIC, thanks to UDP, is really 1 RTT in common cases...
  - or even 0 RTT under conditions

However, do not forget that TCP is not slow on purpose, and that connection-oriented communications have benefits.
Variants from the Happy Path

Version Negotiation

▶ in case the server does not like the client version
▶ the server sends its supported versions in a VersionNegotiation
▶ and the client has to come back
Variants from the Happy Path

Version Negotiation
- in case the server does not like the client version
- the server sends its supported versions in a `VersionNegotiation`
- and the client has to come back

Retry Mechanism
- if the server wants to validate the return path
- it answers with a `Retry` message including a token
- and the client has to come back with the token
Variants from the Happy Path

Version Negotiation

- in case the server does not like the client version
- the server sends its supported versions in a `VersionNegotiation`
- and the client has to come back

Retry Mechanism

- if the server wants to validate the return path
- it answers with a `Retry` message including a token
- and the client has to come back with the token

TLS 1.3 Hello Retry Request

- if the TLS 1.3 `ClientHello` does not contain sufficient information
- the server Initial Packet will contain a TLS 1.3 `HelloRetryRequest`
- and the client has to come back with an updated `ClientHello`
QUIC in a Nutshell

QUIC Main Goals and Features

Performance properties

- low-latency session establishment (1 RTT or even 0 RTT)
- stream multiplexing within a shared connection
- low bandwidth usage (variable length fields)
QUIC Main Goals and Features

Performance properties
- low-latency session establishment (1 RTT or even 0 RTT)
- stream multiplexing within a shared connection
- low bandwidth usage (variable length fields)

Security properties
- state-of-the-art cryptographic primitives
- privacy-oriented measures
- countermeasures against UDP amplification attacks
QUIC Main Goals and Features

Performance properties
- low-latency session establishment (1 RTT or even 0 RTT)
- stream multiplexing within a shared connection
- low bandwidth usage (variable length fields)

Security properties
- state-of-the-art cryptographic primitives
- privacy-oriented measures
- countermeasures against UDP amplification attacks

Compatibility with internet (debatable)
- detailed description of the protocol invariants across versions
- encrypt as much as possible (only parts of the header are in cleartext)
Introduction

QUIC in a Nutshell

QUIC Packet Protection

A Look at QUIC Draft 23 Implementations

Conclusion and Perspectives
A Convoluted Procedure

Unprotected QUIC Packet

Header Payload

Packet Number

Associated Data

Nonce

Key

Plaintext

AEAD (AES-GCM)

Encrypted Payload

header protection key

XOR (selected fields)

AES-ECB

Sampling

Masked Header

Encrypted Payload

Protected QUIC Packet
The Special Case of Initial Packets

Initial Packets are protected, but where do the keys come from?

The initial secret is derived from

▶ a cleartext field in the Client Initial Packet
The Special Case of Initial Packets

Initial Packets are protected, but where do the keys come from?

The initial secret is derived from

- a cleartext field in the Client Initial Packet
- a public value (the salt), depending on the protocol version
The Special Case of Initial Packets

Initial Packets are protected, but where do the keys come from?

The initial secret is derived from

- a cleartext field in the Client Initial Packet
- a public value (the *salt*), depending on the protocol version

Expected benefit from the WG (highly debatable)

- protection against off-path attackers
- robustness against QUIC version-unaware middleboxes
Header Protection Keys

Parts of the Header are also protected

- the hp key is derived from the initial secret
- a mask is generated using the encrypted payload as input
- the hp key stays the same during the whole connection
Header Protection Keys

Parts of the Header are also protected
- the \( hp \) key is derived from the initial secret
- a mask is generated using the encrypted payload as input
- the \( hp \) key stays the same during the whole connection

Expected privacy benefit
- today, the only protected field is the Packet Number
- masking it should help provide unlinkability in case of address migration
Protecting a QUIC packet

1. build the header from its fields
2. build the payload from its fields
3. pad the payload so the packet size is long enough
4. report the payload length in the header to take the padding into account
5. derive secrets and IVs from the version and the DCID
6. derive the nonce from the IV and the Packet Number
7. encrypt the payload
8. extract the sample
9. encrypt the header
Implementation of the Initial Exchange with Scapy (2/2)

The protection procedures mix three types of steps

- classical building/parsing steps
- cryptographic operations
- raw manipulations on the packet

This complexity might lead to subtle bugs in corner cases
- the exact header/payload delimitation is lost during packet protection
- a variable length fields is updated after the initial building phase

We believe this mechanism offers limited benefits (restricted attacker model, cooperating middleboxes) which does not justify the induced complexity.
Implementation of the Initial Exchange with Scapy (2/2)

The protection procedures mix three types of steps

- classical building/parsing steps
- cryptographic operations
- raw manipulations on the packet

This complexity might lead to subtle bugs in corner cases

- the exact header/payload delimitation is lost during packet protection
- a variable length fields is updated after the initial building phase
Implementation of the Initial Exchange with Scapy (2/2)

The protection procedures mix three types of steps

- classical building/parsing steps
- cryptographic operations
- raw manipulations on the packet

This complexity might lead to subtle bugs in corner cases

- the exact header/payload delimitation is lost during packet protection
- a variable length fields is updated after the initial building phase

We believe this mechanism offers limited benefits (restricted attacker model, cooperating middleboxes) which does not justify the induced complexity
Test Servers

In the QUIC WG wiki, existing implementations are listed

- 16 different stacks are listed
- corresponding to 20 public servers

We led measurement campaigns (related to different draft versions)

- several servers never answered any stimuli
- others had significant down times, especially after a new draft version
- around 10-12 seem to keep up with the latest draft
Test Servers

In the QUIC WG wiki, existing implementations are listed
- 16 different stacks are listed
- corresponding to 20 public servers

We led measurement campaigns (related to different draft versions)
- several servers never answered any stimuli
- others had significant down times, especially after a new draft version
- around 10-12 seem to keep up with the latest draft

Warning: the presented results are partial data on still evolving implementations
Version Negotiation

Stimuli

1. a valid Initial Packet with a supported draft version
2. packet 1 with a yet-to-be defined version
3. a truncated version of packet 2

Expected result

- the first packet should be accepted
- the second and third packet should trigger a VersionNegotiation
Version Negotiation

Stimuli

1. a valid Initial Packet with a supported draft version
2. packet 1 with a yet-to-be defined version
3. a truncated version of packet 2

Expected result

- the first packet should be accepted
- the second and third packet should trigger a VersionNegotiation

Actual result

Several servers choke on the third packet, which shows that they interpret the packet length field, although this field could be redefined in the future (cf. draft-quic-invariants)
Client Initial Packet Length

To limit DoS amplification attacks, QUIC states that

- the Client Initial Packet should at least be 1,200 bytes long
- before the Handshake is complete, the server should not answer with more than 3 times the amount received
Client Initial Packet Length

To limit DoS amplification attacks, QUIC states that

- the Client Initial Packet should at least be 1,200 bytes long
- before the Handshake is complete, the server should not answer with more than 3 times the amount received

Observations

- several servers accept 300-byte long stimuli
- but only answer with up to 900 bytes

This is not ideal, nor dramatic.
Missing Parameters

The specification contains several requirements about TLS 1.3 extensions, including these ones:

- ALPN is mandatory
- QUIC Transport Parameters must be sent
Missing Parameters

The specification contains several requirements about TLS 1.3 extensions, including these ones

- ALPN is mandatory
- QUIC Transport Parameters must be sent

Deviations

- the sample packet in the draft does not conform to the requirements
- several implementations accommodate missing extensions
- one implementation only accepted our stimuli without ALPN
Frame Mangling

Initial Packets should only contain
- Crypto frames (and the ClientHello should not be split)
- ACKs
- Padding frames
- Connection Close messages

However, several servers seem to accept
Frame Mangling

Initial Packets should only contain

- Crypto frames (and the ClientHello should not be split)
- ACKs
- Padding frames
- Connection Close messages

However, several servers seem to accept

- Ping frames (allowed in draft-24)
Frame Mangling

Initial Packets should only contain

- Crypto frames (and the ClientHello should not be split)
- ACKs
- Padding frames
- Connection Close messages

However, several servers seem to accept

- Ping frames (allowed in draft-24)
- a ClientHello split into two frames (draft-24 allows spanning over several packets)
Frame Mangling

Initial Packets should only contain

- Crypto frames (and the ClientHello should not be split)
- ACKs
- Padding frames
- Connection Close messages

However, several servers seem to accept

- Ping frames (allowed in draft-24)
- a ClientHello split into two frames (draft-24 allows spanning over several packets)
- a Crypto frame split into two overlapping frames
Frame Mangling

Initial Packets should only contain

- Crypto frames (and the ClientHello should not be split)
- ACKs
- Padding frames
- Connection Close messages

However, several servers seem to accept

- Ping frames (allowed in draft-24)
- a ClientHello split into two frames (draft-24 allows spanning over several packets)
- a Crypto frame split into two overlapping frames
- and even a Crypto frame inconsistently split!
Introduction

QUIC in a Nutshell

QUIC Packet Protection

A Look at QUIC Draft 23 Implementations

Conclusion and Perspectives
Conclusion

▶ QUIC is a protocol still under development
▶ It is worth studying, since it could become an important part of the web traffic
▶ It is a complex beast

From the implementation point of view
▶ we wrote a first implementation of the protocol in Scapy
▶ we scanned public servers with corner case stimuli
▶ no server seems to conform to all the requirements we looked at
▶ however, these stacks are fast-evolving implementations of a moving target
Future work

Regarding our Scapy implementation

▶ stabilize a version against the last drafts
▶ publish the code
▶ include other features (0 RTT, address migration)

Regarding the IETF WG and the ecosystem

▶ contribute to discussions on the WG list
▶ include our test suite in existing tools such as QUIC Tracker

Other (GASP) ideas

▶ try and implement QUIC specs with our tools
▶ fuzz the implementations (packets and state machines)

Possible collaborations (or internships) if you (or your students) are interested
Future work

Regarding our Scapy implementation

► stabilize a version against the last drafts
► publish the code
► include other features (0 RTT, address migration)

Regarding the IETF WG and the ecosystem

► contribute to discussions on the WG list
► include our test suite in existing tools such as QUIC Tracker
Future work

Regarding our Scapy implementation
▶ stabilize a version against the last drafts
▶ publish the code
▶ include other features (0 RTT, address migration)

Regarding the IETF WG and the ecosystem
▶ contribute to discussions on the WG list
▶ include our test suite in existing tools such as QUIC Tracker

Other (GASP) ideas
▶ try and implement QUIC specs with our tools
▶ fuzz the implementations (packets and state machines)
Future work

Regarding our Scapy implementation
- stabilize a version against the last drafts
- publish the code
- include other features (0 RTT, address migration)

Regarding the IETF WG and the ecosystem
- contribute to discussions on the WG list
- include our test suite in existing tools such as QUIC Tracker

Other (GASP) ideas
- try and implement QUIC specs with our tools
- fuzz the implementations (packets and state machines)

Possible collaborations (or internships) if you (or your students) are interested
Questions?

Thank you for your attention

@pictyeye
olivier.levillain@telecom-sudparis.eu
https://paperstreet.picty.org/yeye