Semester project: Encrypted R2P2

Guillaume Solignac - supervised by Marios Kogias
EPFL DCSL - Spring 2020

***Abstract*— The following document presents my Semester project. It aims at providing end-to-end encryption on top of R2P2 [3], a datacenter-oriented transport protocol tailored for RPCs.**

I. Introduction

In the recent years, datacenters have become a crucial element in the development of *cloud computing*. More and more companies choose to use cloud providers for part or totality of their IT infrastructure. Performance and security are two fundamental issues that need to be addressed inside datacenters. In this paper, we focus on RPCs.

A. Datacenter performance

Inside datacenters, popular RPCs frameworks are usually set up on top of TCP (typically DCTCP). However, networking inside datacenters has different constraints than regular Internet networking. To avoid some known limitations of TCP in this context, other alternatives have emerged, such as R2P2 (cf. II-A).

B. Datacenter security

However, using *cloud*-based solutions has specific security implications. [1] has done an extensive work of synthesizing various aspects. This section presents the relevant parts for our work.

a) Multi-tenancy: There are various possibilities for the backend of a cloud, and common options are *public clouds* and *community clouds*. They imply sharing the infrastructure with other parties, either other clients or other partners. Basically, it means that there can be rogue devices on the datacenter network, even they should be isolated from each other.

b) Networking: While managing a network is already prone to errors, networking in a datacenter raises even more concerns. Misconfigurations can weaken network isolation, which has even more impact when there are multiple tenants on a facility.

c) Physical security: Physical facilities are exposed to potentially malicious agents (eg. a rogue employee plugging a malicious device on a router).

The performance issue was the main focus of the initial version R2P2, and this work aims at adding encryption on top of it to tackle the security issue.

II. Background

A. R2P2 Background

R2P2 [3] is a transport-level protocol designed to perform high-performance RPCs inside datacenters. It achieves high performance by separating target selection from data flow. Fig. 1 shows a typical R2P2 flow : target selection is performed by a middlebox, and the rest of the interaction is done directly between the endpoints. This limits the traffic going through the middlebox, thus avoiding to create a bottleneck there.

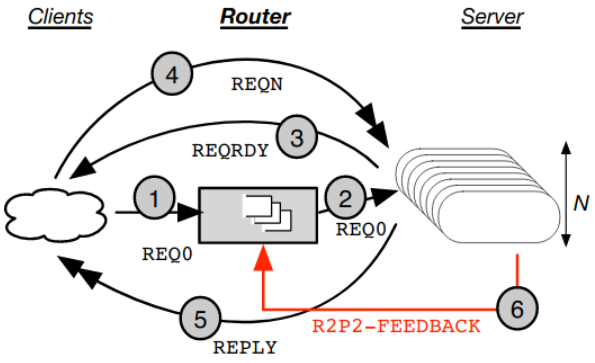

Fig. 1. R2P2 protocol overview. Reproduced from [3].

B. TLS Background

1) Overview: Transport Layer Security (TLS) [4] is a protocol used to provide a secure channel (with confidentiality and integrity) between two

endpoints, typically a client and a server. It uses certificates so that a client is able to authenticate a server with the help of a third party Certificate Authority (CA).

2) Handhsake: A TLS Session starts with a handshake. During the exchange, both endpoints exchange keys and the client checks the server authenticity. Once session keys are established, data going through the session is protected.

The first message is the TLSClientHello. It contains a list of encryption algorithms supported by the client, a random nonce used to derive keys, and eventually a PreShared-Key (PSK) label used to agree on a PSK.

The server then replies with a TLSServerHello which contains the choosen cryptographic parameters, a random nonce used to derive keys, the certificate and a proof that the server has the private key associated with the certificate. It derives its session keys either from the client random nonce and its own random nonce or from the PSK if a PSK label was present.

The client can verify the server authenticity and derive the same session keys, then proceeds to send encrypted data.

3) Stateless Resumption and early data: TLS offers two features that we used to build our system. The first one is stateless resumption (4.6.1 of [4]): during the first connection, the server can provide the client with a ticket (Fig. 2) that can be used as a Pre-Shared Key (PSK) in a future connection. This ticket is encrypted and

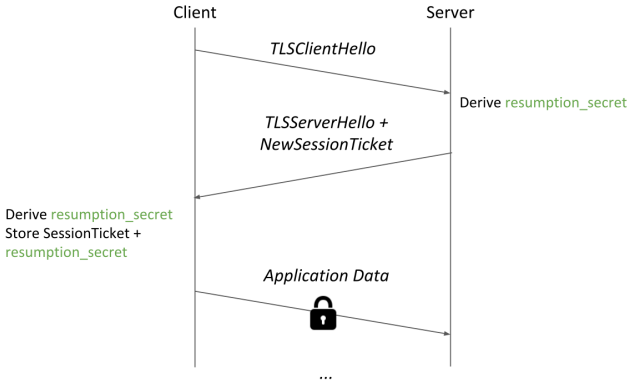

Fig. 2. 1-RTT TLS handshake with NewSessionTicket

authenticated with the Session Ticket Encryption Key (STEK) only known to the server. It contains the resumption key of the connection (also stored by the client), so that the server is able to resume the session with this key.

The other feature is early data. When used with a PSK, the client can send data before the first server replies, removing the first RTT before being able to send data securely.

A combination of both features can be seen Fig. 3.

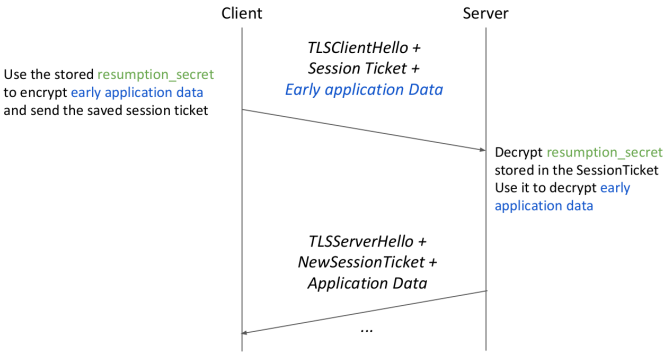

Fig. 3. 0-RTT TLS handshake and early data

4) Security: TLS brings security against both active and passive network attackers.

a) Forward secrecy: TLS also provides *forward secrecy*, which means that even if long-term keys are compromised, records are still confidential provided the keys for that session are deleted. In TLS, this is achieved through Diffie-Hellman negotiation on every session establishment. However, when using PSK, DH negotiation is optional, which can cancel forward secrecy.

b) Replay attacks: Early data is not protected against replay attacks. It means that resending a valid packet with early data will be accepted by the server if no anti-replay mechanism is set up.

III. Encrypted R2P2

In this part, we explain the design choices that led to encrypted R2P2. [2] shows a real-world scenario using TLS for RPCs over TCP, and some design ideas were reused for this work.

A. Design

The main requirement of the design was to keep changes to a minimum to maintain the performance gains of R2P2 to a maximum.

There are two cases:

1) Single packet requests: For single packet requests, adding an RTT to perform the TLS handshake would cost a lot latency-wise: the two TLS features described II-B.3 are used to avoid this. If a ticket for the service is available, it is used to send the request as early data. If the ticket is not valid (which should never happen), the server discards early data then proceeds with a regular handshake.
If no ticket is available (which should happen only once), a regular handshake is done, which will eventually get a new ticket.

2) Multiple packet requests: In this case, there is only a small difference. The first packet remains a simple R2P2 packet, along with the TLS Ticket and early data. The only difference is that the client cannot pre-compute the number of total packets before the handshake is done. To overcome this, the R2P2 sequence number of the first packet takes the special value -1. The next packet contains the number of remaining packets to be sent.

In this design, all services get a name which is used inside the SNI field of the certificate. All servers serving a service get a valid certificate.

In typical deployment setups, many servers might receive various requests from a single host. However, even if all servers get a valid certificate, this does not mean they can resume a session established with an arbitrary server. To make the system work well, the STEK must be the same among all the servers. It means a ticket is valid for every server. If not, almost every early data would be discarded which would kill performance drastically.

B. Implementation

The implementation is available at https://github.com/gsol10/r2p2/tree/encrypted.

1) API changes: Some API changes were needed to include TLS. When an R2P2 client sent a request, it needed to know the target IP address: now it needs a service name which the server certificate will be checked against.

API Call	Description
`r2p2_send_req`	Added an argument to specify the service name to check.
`r2p2_tls_init`	An initialisation call for TLS. Needs to be called one at startup, both on server and client side.
`r2p2_set_stek`	Used to set the current key.

2) PicoTLS: The prototype was built on top of PicoTLS [1], which was originally created to implement TLS for HTTP/2 on top of Quic. It provides a flexible API that made the implementation straghtforward. The library aims to implement all TLS 1.3 features, which makes it easy for future changes of the implementation.

3) Tickets management: In the Linux implementation, sockets are bound to a thread, so having one ticket per thread (and per Server Name) is suitable: there are no concurrency problems. Under heavy load, a new ticket is available every time the server acknowledges a new request: tickets will change regularly, so there should not be any tickets discarded because of ticket time validity.

C. Security

By using TLS, Encrypted R2P2 protects against passive and active (except for replay attacks in the current version) attackers. Given that other RPC systems also use TLS, it makes R2P2 security comparable to them.

However, there is one crucial difference : sharing the STEK across servers means that a single server takedown breaks the confidentiality of all other exchanges with other servers, which would not be the case with DH negotiation on every R2P2 request.

1) Forward secrecy: As pointed in II-B.4, TLS without DH negotiation looses forward secrecy. This is due to the fact that every session initiated with a ticket encrypted with a given STEK is decipherable with this STEK.

To mitigate this issue, STEKs must be rotated frequently enough. Past STEKs must be erased from every system to avoid future compromises. [2] proposed a way to achieve graceful STEK

[1] https://github.com/h2o/picotls

	Early data	CPU consumption	Single-node takedown resilience	Practical deployment
STEK	Yes	Low	Breaks the confidentiality for the whole service	STEKs to share.
DH negotation	No	High	Confidentiality of exchanges with other servers is maintained	Nothing more than the certificate
PSK	Yes	Low	Breaks the confidentiality for the whole service	PSK needs to be synchronized among servers and clients

Fig. 4. Comparison of the available options for TLS deployment for R2P2.

rotation without having loads of invalid tickets during transitions. This demonstrate the operational feasibility of using STEKs.

Another option would be to enable DH negotiation on every R2P2 request, which would avoid to share the STEK across servers. This would drastically increase CPU consumption, but benchmarks should be run to tell if it is a possible option on a case by case basis.

2) Replay attacks: The use of early data for the first packet of an R2P2 RPC makes the system prone to replay attacks. Users must carefully check that this does not represent a security issue for them (refer to E.5. of [4] for a comprehensive list of the risks). If the use of early data is not possible, it can be disabled server-side.

It is highly possible that this becomes an important issue. Anti-replay mechanisms could be integrated to R2P2. One possibility comes to mind: including the R2P2 request ID and sender IP address inside the integrity protected part of the request, so that servers can keep a shared state of already used request IDs and discard them if needed. To avoid sharing the state between servers, which could rapidly become a practical nightmare, the receiver IP address can also be included.

Fig. 4 sums up the security and practical differences between the different options: STEK refers to the proposed design, DH negotiation means no tickets used, and PSK refers to a scenario where a pre-shared key is used.

D. Performance

1) Benchmarks: In this part, we compare the unloaded performance of R2P2 and encrypted R2P2. We also compare it with gRPC and gRPC over TLS. For this, we create an echo server that receives an RPC and replies with the same payload. The server is then run and a client sequentially sends 10000 4-bytes requests. The completion time is measured for each RPC. Branches BENCH of the repository contain the benchmarking applications.

Fig. 5 shows the results of the experiments.

First, the completion time is about 7 times longer with encrypted R2P2. Second, encrypted R2P2 is still faster than gRPC, and about 1.8 times faster than gRPC over TLS.

The other aspect is that there is a larger standard deviation with gRPC than with R2P2. R2P2 seems more stable than gRPC.

2) Overhead sources: III-D.1 showed that the performance overhead of encrypted R2P2 is fairly high. Some possible sources for this overhead are:

- Encryption times. These should be fairly low. AES-NI enabled machines can achieve around 1GB/s, so in the experiment the price should be almost insignificant.
- TLS machinery. Establishing a session, sending and receiving data involves a number of mandatory operations for the TLS library. Profiling code is necessary to assess the impact of this.
- Unoptimized code. The prototype used for encrypted R2P2 has a number of places prone to bad performance, especially in the realm of memory management for TLS sessions. Again, profiling code is necessary to see if it is the main source of overhead for encrypted

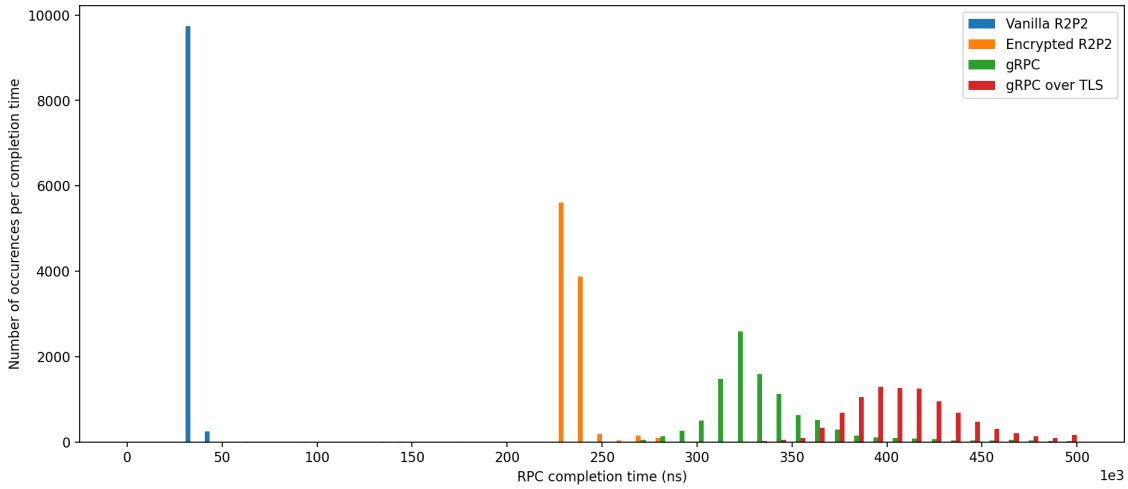

Fig. 5. Histograms of RPC completion times for different transport protocols. Experiments consisted on 10000 consecutive RPC calls on a echo service with 4 bytes payloads, between one client and one server. Average values : vanilla R2P2 $34\mu s$, encrypted R2P2 $237\mu s$, vanilla gRPC $336\mu s$, encrypted gRPC $424\mu s$.

R2P2. It would be the best case scenario, as it would allow for optimizations and better performance.

Another overhead source not captured by the experiments is the byte overhead. Network captures show that around 300 bytes are added to the first message of the client which contains TLS parameters and the TLS ticket (around 100 bytes). The first message of the server, containing the new TLS ticket, is composed of around 300 bytes of TLS data as well.

IV. Conclusion

Using TLS allowed to make R2P2 state-of-the-art security-wise. It also allows for straightforward future changes using the large number of TLS features. Changes to the original protocol were made so that the total performance overhead would be as limited as possible. Benchmarks looks promising: even though they show a large overhead compared to vanilla R2P2, Encrypted R22P still beats gRPC. The prototype implementation should be profiled to see if these overheads are inherent to the design or if they are just a byproduct of a poor implementation.

References

[1] Diogo A. B. Fernandes, Liliana F. B. Soares, João V. Gomes, Mário M. Freire, and Pedro R. M. Inácio. Security issues in cloud environments: a survey. *International Journal of Information Security*, 13(2):113–170, Apr 2014. doi:10.1007/s10207-013-0208-7.

[2] Neel Goyal, Kyle Nekritz, and Subodh Iyengar. Building facebook's service encryption infrastructure. URL: https://engineering.fb.com/security/service-encryption/.

[3] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/kogias-r2p2.

[4] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, August 2018. URL: https://tools.ietf.org/html/rfc8446.