Implementing the Plugin Distribution System

Nicolas Rybowski, Quentin De Coninck, Tom Rousseaux, Axel Legay, Olivier Bonaventure
UCLouvain, Belgium
firstname.lastname@uclouvain.be

ABSTRACT

Recent works proposed to dynamically extend protocol implementations through protocol plugins. While addressing deployment issues, they raise safety concerns (do they terminate, do they act maliciously,…). To fill this gap, a system distributing trust in plugin's verification properties was proposed in the literature. However, it was not implemented. This poster demonstrates the feasibility of this approach by providing an open-source implementation of this system. We also extend the state-of-the-art verification works about protocol plugins by considering a new property called side-effects.

CCS CONCEPTS

• **Networks** → **Protocol testing and verification**;

KEYWORDS

Distributed verification system, Plugin, Protocol operation, Safety properties, PQUIC

ACM Reference Format:
Nicolas Rybowski, Quentin De Coninck, Tom Rousseaux, Axel Legay, Olivier Bonaventure. 2021. Implementing the Plugin Distribution System. In *SIGCOMM '21 Poster and Demo Sessions (SIGCOMM '21 Demos and Posters), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3472716.3472860

1 INTRODUCTION

Important Internet protocols are designed to be easily extensible to cope with new application requirements. However, implementations evolve at a slow pace and extensions often require support from all participating entities. Researchers proposed pluginized protocols (PQUIC [1], xBGP [5], …) as an alternative architecture. Base implementations are deployed once and can be dynamically extended through machine-independent bytecode called protocol plugins. Anyone can develop these plugins and distribute them by a side-channel or through the pluginized protocol itself.

While plugins enable extension deployments from day one, their untrusted nature raises safety issues. To address them, PQUIC's authors [1] proposed a distributed verification system called Plugin Distribution System (PDS) where entities can assess that a given plugin follows safety rules. In a nutshell, developers can distribute their plugins at large scale through Plugin Repositories (PRs). Plugin Validators (PVs) can register on these PRs to verify these plugins. If a plugin satisfies the PV safety requirements, the PV includes it in a Merkle tree and computes its Signed Tree Root (STR). Any host can check whether a given plugin has been verified by a PV by reconstructing the PV's Merkle tree using hash functions. If the check ends with the value of the PV's STR, then it means the PV validated the plugin. Any host can fetch STRs either through PRs or by directly contacting the PVs. This system offloads the verification tasks from the pluginized hosts. Hence, it allows verifying complex properties with time and computation intensive tools.

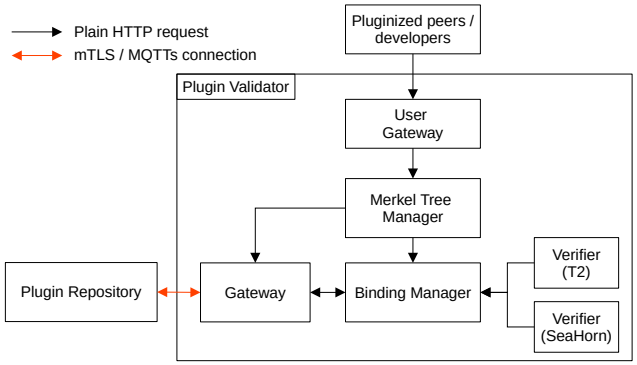


Figure 1: Components of a Plugin Validator.

However, this Plugin Distribution System was not implemented and it remained unclear if such a system would be feasible in practice. Furthermore, PQUIC's authors only checked that plugins always terminate. Malicious plugins may still advertise a given functionality while actually exposing a different behavior, such as accessing or modifying unrelated connection's fields.

In this poster, we propose a modular, open-source implementation of the Plugin Distribution System [4]. In particular, we implement a generic Plugin Validator and a basic Plugin Repository. Our implementation enables the inclusion of several verification tools inside a single PV. We demonstrate this by using SeaHorn [2] to verify plugins altering connection fields. The source-code of this tool is also publicly available [3].

2 IMPLEMENTATION

Our Plugin Distribution System (PDS) consists in a set of PVs and PRs. Developers push their plugins to PRs that can then distribute them to all existing PVs. To handle such events, we rely on a Publish-Subscribe messaging pattern using the MQTT protocol, allowing PRs to reach all subscribed PVs without being aware of them. We rely on mutual TLS with Public Key Infrastructure for the communications that require authentication, e.g., the ones between PR and PV. Our PDS deploys its own root Certificate Authority which is used to sign certificates for PVs, PRs and developers. These certificates are then used for secure MQTTs and classical TLS connections between the PVs and the PRs.

SIGCOMM '21 Demos and Posters, August 23–27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8629-6/21/08…$15.00
https://doi.org/10.1145/3472716.3472860

Each PDS component is implemented as a Kernel Virtual Machine (KVM) running a stack of micro-services. Figure 1 illustrates our overall PV implementation architecture. It consists in two main components: (*i*) the micro-services forming its core and (*ii*) the embedded verifiers. The core is common to all PVs and is protocol agnostic, while verifiers are property and/or protocol specific. They implement a simple routine which queries, at regular interval, the core for unverified plugins. Each new plugin is verified and the result of the verification is sent back to the core. Each core service presents a REST API for inner PV communication.

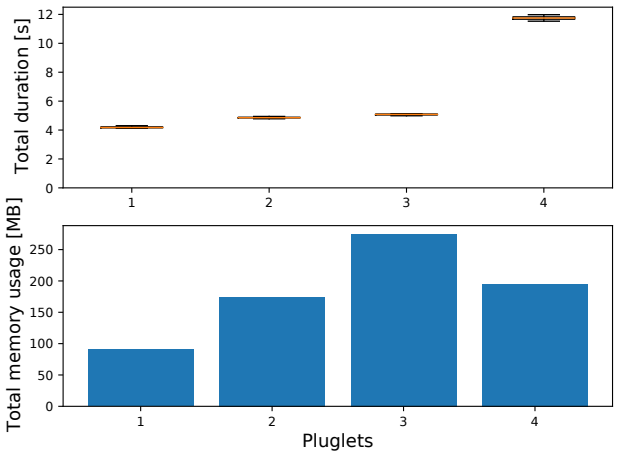
Our PV architecture uses two different gateways isolating the PV core from the outside world. The first one focuses on communications with PRs. It is the only way to modify the inner PV state from the external world. Hence, their exchanges require authentication from both entities. The second (user) gateway is publicly accessible and serves the PV's signed Merkle tree root. The Binding manager stores all known information (source code, bytecode, manifest, ...) on every plugin processed by the PV. It serves plugin's data to verifiers and stores their verification results. The Merkle tree manager produces the Merkle tree at regular intervals and provides access to the data stored within. The Merkle tree includes plugins succeeding all verifiers. Its depth is statically defined at PV deployment time. Considering a Merkle tree of depth 7 containing 16 plugins, our prototype creates it in about 2.8 ms and pluginized hosts takes 0.839 ms to check with the STR that a plugin is inside it. This duration comprises both the root hash computation (0.142 ms) and the STR validation (0.697 ms). The STR contains the root hash of the PV's Merkle tree, signed with the PV's private key. Hence, the STR validation corresponds to the comparison of the root hash recomputed by the pluginized host and the one contained in the STR (extracted using the PV's public key).

The whole PDS prototype is about 845 lines of code, excluding the code required for its deployment.

In the current prototype version, the PR might be a single point of failure. However, future works envision an interconnection of multiple PRs ensuring the whole system availability. The PVs are, by nature, not single points of failures since they do not have to be individually trusted. Depending on its safety requirements, a pluginized peer may query multiples PVs to enforce its trust in a plugin verification status. Also, even whether they are extensively tested before being integrated in a PV, the verifiers can make mistakes or not being able to prove a given property. Again, it is up to the pluginized peers to choose which PVs they trust by operating a form of consensus on their outputs. Eventually, the whole PDS should converge to a valid result.

3 VERIFYING PROTOCOL PLUGINS

Protocols plugins introduce a new programming model where an implementation executes bytecode within an eBPF virtual machine. A specific safety property is *side-effects*, i.e., which session fields are altered by a given plugin. The plugin manifest contains a high-level specification listing the session fields whose value can be changed after its execution. To verify this property, we extend the source code of each plugin function as follows. First, we initialize a context and inputs for the function's call. Then, we call the plugin's function. Finally, we use SeaHorn assertions to verify that the PQUIC context,



1 `be.michelfra.disable_cc/congestion_algorithm_notify` (12 SLOC)
2 `be.michelfra.westwood/congestion_algorithm_notify` (122 SLOC)
3 `be.mpiraux.ack_delay/is_ack_needed` (27 SLOC)
4 `be.mpiraux.ack_delay/update_ack_delay/` (21 SLOC)

Figure 2: Performance measurements of pluglets verification of their side-effects property.

except the fields listed in the manifest, remains unchanged. This verification is performed statically with formal methods.

We consider the plugins publicly available on the PQUIC's repository. Among them, we succeed to verify the side-effects propertie of three plugins consisting of four different plugin's functions. These plugins consists in congestion control algorithms or adaptation of the delay for ACK frame generation.

Figure 2 illustrates the resource usage of their verification on an Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz (4 cores, 8 threads) CPU with 16 GB of DDR3 1600MHz RAM. This verification process takes up to a dozen of seconds, which is reasonable for the offline approach proposed by the PDS. The time variation of those measures is probably due to the structure of the data manipulated by those plugins. For instance, pluglet 4 takes a non-deterministically bounded linked list as argument. However, the online approach, i.e., letting pluginized hosts verify plugins on-the-fly just before injecting them on connections, remains unpractical. The memory usage could also be problematic on memory-limited hosts. This shows that the PDS is relevant to perform complex verification.

Our preliminary results show that it is possible to verify other properties than termination of protocol plugins. Yet, there remain areas of improvements for this. We also considered 12 other plugin's functions (from 7 different plugins), but their verification fails. Those failures have several origins. For some of them, workarounds are possible by slightly modifying the plugin's code, e.g. changing the model of some PQUIC memory accesses. This was done on the *congestion_algorithm_notify* implementation of the Westwood plugin. Our future work will extend those fixes to the failing plugin's functions. For the other issues, some adaptations are required in the PQUIC model used for the verification. For instance, we will reconsider plugins calling other plugins or functions external to the PQUIC API, and the usage of a formal specification of this API.

REFERENCES

[1] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing quic. In *Proceedings of the ACM Special Interest Group on Data Communication*. 59–74.

[2] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.

[3] Nicolas Rybowski. 2021. Side-effects verification tool for PQUIC plugins. https://github.com/nrybowski/pquic-formal-model. (2021).

[4] Nicolas Rybowski. 2021. SPMS prototype implementation. https://github.com/nrybowski/SPMS. (2021).

[5] Thomas Wirtgen, Quentin De Coninck, Randy Bush, Laurent Vanbever, and Olivier Bonaventure. 2020. xBGP: When You Can't Wait for the IETF and Vendors. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 1–7.