Boire, Sebastien; Akgün, Tolgahan; Ginzboorg, Philip; Laitinen, Pekka; Tamrakar, Sandeep; Aura, Tuomas

Credential Provisioning and Device Configuration with EAP

Published in:
MobiWac '21: Proceedings of the 19th ACM International Symposium on Mobility Management and Wireless Access

DOI:
10.1145/3479241.3486705

Published: 22/11/2021

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
Boire, S., Akgün, T., Ginzboorg, P., Laitinen, P., Tamrakar, S., & Aura, T. (2021). Credential Provisioning and Device Configuration with EAP. In *MobiWac '21: Proceedings of the 19th ACM International Symposium on Mobility Management and Wireless Access* (pp. 87–96). ACM. https://doi.org/10.1145/3479241.3486705

Credential Provisioning and Device Configuration with EAP

Sebastien Boire
Aalto University, Finland
boire.sebastien@gmail.com

Tolgahan Akgün*
F-Secure, Finland
akguntolgahan@gmail.com

Philip Ginzboorg
Huawei, Finland
philip.ginzboorg@huawei.com

Pekka Laitinen
Huawei, Finland
pekka.laitinen@huawei.com

Sandeep Tamrakar
Huawei, Finland
sandeep.tamrakar@huawei.com

Tuomas Aura
Aalto University, Finland
tuomas.aura@aalto.fi

ABSTRACT

The Extensible Authentication Protocol (EAP) is used for authenticating client devices to WiFi networks, and it is designed to be extensible with new authentication methods. We look at ways to extend the protocol to support credential provisioning and configuration of new client devices. As large numbers of IoT devices are deployed, the task will be simplified by combining the network connectivity, identity and certificate provisioning, and application-layer connectivity to one process. The solution will also allow the use of a one-time credential for the initial authentication, so that the long-term device certificate is issued automatically after the first connection to the network. The paper analyzes the requirements and architectural design options that implement such a user experience. We consider solutions that transfer short bootstrapping data inside the EAP session and then implement the provisioning and configuration with web APIs over HTTPS. This allows future flexibility and speed of development in the provisioning and configuration steps. We designed and implemented several architecturally different solutions and present the comparison results and also compare with previous proposals that have similar goals.

CCS CONCEPTS

• **Networks** → **Network protocol design**; • **Security and privacy** → **Mobile and wireless security**.

KEYWORDS

Wireless network, security, EAP, device management, certificate provisioning, configuration

ACM Reference Format:
Sebastien Boire, Tolgahan Akgün, Philip Ginzboorg, Pekka Laitinen, Sandeep Tamrakar, and Tuomas Aura. 2021. Credential Provisioning and Device Configuration with EAP. In *Proceedings of the 19th ACM International Symposium on Mobility Management (MobiWac '21), November 22–26, 2021, Alicante, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3479241.3486705

*The work was done while at Huawei, Finland.

MobiWac '21, November 22–26, 2021, Alicante, Spain.

ACM ISBN 978-1-4503-9079-8/21/11.
https://doi.org/10.1145/3479241.3486705

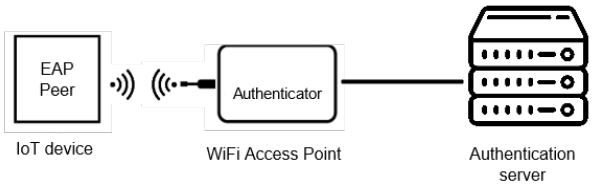

Figure 1: Entities participating in EAP session.

1 INTRODUCTION

This paper investigates how to extend the Extensible Authentication Protocol (EAP) [21], which is typically used for authenticating wireless network clients in order to grant them network access, with features for managing the client devices. Device management operations include provisioning of the device's long-term credentials and configuration of the device, i.e., setting of device parameters, as well as installation of software updates. These operations take place when the device joins its home network for the first time.

EAP is a framework for authenticating network clients. It provides a uniform way to integrate existing authenticated key-exchange mechanisms to network access authentication. The most common application of EAP is WPA-Enterprise in WiFi networks. An *EAP method* is defined for each authentication mechanism, e.g., EAP-TLS for the TLS protocol. The authentication takes place between the network client, called the *peer* or *supplicant*, and an *authentication server* on the backend network (see Figure 1). The network access server, which is typically the WiFi access point, is called the *authenticator* in EAP. It conveys EAP messages between peer and authentication server. If the authentication succeeds, the server informs the authenticator, which grants network access to the peer device. Tens of different EAP methods exist, and EAP is widely adopted in enterprise wireless networks. Some EAP methods consist of two layers: the outer or wrapper method typically authenticates the server with its TLS certificate while the inner method authenticates the user, for example, with a username and password.

The Internet of Things (IoT) consist of large numbers of network-connected smart objects, ranging from simple sensors to home appliances and industrial machinery. The growing number and diversity of the connected devices within both homes and enterprises makes their management a challenging task. It becomes important to manage the networks access for individual devices. Devices that are not fully trusted should be isolated from others on the network, and when a device is decommissioned or lost, its access to the network should be revoked. This means that the current security

mechanisms, such as shared WiFi passphrase or authentication with the user's personal credentials, are no longer acceptable. Instead, each device needs to have an individual identity and access credentials.

EAP authentication with device-specific credentials is the readily available standard solution for authenticating the individual devices. However, a second problem arises: the manual configuration of each device with vendor-specific methods becomes tedious. In order for this solution be deployable, the device configuration and credential provisioning for network access need to be automated and streamlined in a vendor-independent way. This paper investigates the different ways to integrate the automated provisioning and configuration step to EAP. The idea is to bootstrap the network connectivity with any easily available method and then automatically issue the device-specific credentials for long-term use.

Two existing EAP methods include an optional step for provisioning long-term credentials to the device. In both, the provisioning is protected by a TLS tunnel. Tunnel Extensible Authentication Protocol (EAP-TEAP) [22] includes a sub-protocol for optionally issuing a certificate to the peer. Credentials Provisioning and Management via EAP (EAP-CREDS) [13] is a draft proposal for running several provisioning protocols as an inner EAP method inside the TLS wrapper. Both EAP-TEAP and EAP-CREDS provision the long-term credentials within the EAP session. This means that the provisioning protocol becomes a part of the EAP implementation in the network stack. These methods also need to support fragmentation of the provisioning messages and retry in case the process fails.

We consider a slightly different approach: Only send short *bootstrapping data* from the authentication server to the peer within the EAP session, and run the actual provisioning protocol over HTTPS after the EAP authenticator (i.e., WiFi access point) has granted network access to the peer device. The bootstrapping data includes a management server address and a client token, which the client will use to authenticate itself to the server. The access point can initially restrict the device's access to the network using standard techniques [5, 12], so that it is only able to connect to the management server. Full network access can be granted after the device re-authenticates with the newly-provisioned device-specific credentials.

One advantage of our approach is the ease of implementation. It is easier and faster to implement the credential provisioning and device configuration on the application layer over web protocols and REST APIs, compared to coding them in C as part of EAP. Moreover, several IoT platforms already use REST APIs for device management [2, 8]. Another advantage of this approach is that it adds flexibility to the choice of the credential provisioning and configuration methods. There is no need to standardize the full provisioning and configuration process inside EAP.

The contributions of this paper are the following:

- We design and implement the bootstrapping framework as described above.
- We identify several (4) architecturally different technical solutions for extending the EAP protocol to transfer the short bootstrapping data within the EAP session. We implement, analyze and evaluate these different solutions experimentally. Based on the evaluation, we arrive at a preferred solution.
- We compare the preferred solution with the previously existing provisioning technologies.

We implemented the framework using standard web technologies and extended the open-source *hostapd* and *wpa_supplicant* software. Raspberry Pi was used as the hardware platform in the experiments.

The rest of the paper is organized as follows. Section 2 provides background information. In Section 3, we list requirements for the system. In Section 4, we describe the system architecture, the bootstrapping process, and four different options for transferring bootstrapping data in an EAP session. Section 5 provides the details of their implementation. Section 6 contains a summary of evaluation results, which is followed by the discussion in section 7. Finally, Section 8 concludes the paper.

2 BACKGROUND

2.1 Cloud-based device management solutions

The are many ongoing efforts to develop and deploy solutions for managing the growing numbers of network-connected smart devices. Many of them have management servers in the cloud. Samsung SmartThings, AWS IoT, Arm Pelion are some of the proprietary cloud-based solutions. There are also hub devices aimed primarily at home networks, such as Samsung Smart Hub and Google Nest Hub, which provide local provisioning and management features and mediate the connection to the cloud. On the standards side, the Lightweight machine-to-machine (LwM2M) [2] protocol by Open Mobile Alliance, FIDO Device Onboard Specification by FIDO Alliance [6], and Open Connectivity Framework Specifications by Open Connectivity Foundation [7] are attempts at standardizing IoT device management. These specifications were designed to suit the need of resource-constrained IoT devices and provide protocols for bootstrapping, management, and control of IoT devices.

2.2 Extensible Authentication Protocol (EAP)

Already briefly introduced, EAP [21] is a generic framework for transporting messages of different authentication methods, such as TLS or 3GPP AKA, which are called EAP *methods*. EAP is a request-response protocol where the authentication server always sends a request to the peer (client) and the peer returns a response. Each method specification defines how the authentication protocol messages are sent over this communication pattern. EAP can operate directly on top of a wireless LAN, such as 802.11, or Point-to-Point (PPP) protocol [20] without needing the IP layer. EAP is used in wireless networks, and sometimes in wired ones, to authenticate the client before granting it network access.

EAP does not natively support message fragmentation. That is, EAP messages must fit into the network-specific maximum transfer unit (MTU), which according to EAP specification [21], should be at least 1020 bytes. However, some EAP methods, implement fragmentation of longer messages into multiple frames. Methods that make use of the TLS handshake for authentication must support fragmentation to be able to carry the handshake messages.

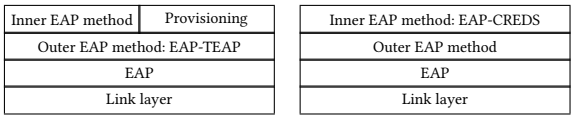

Figure 2: EAP-TEAP and EAP-CREDS protocol stacks

2.3 EAP-TEAP

Tunneled Extensible Authentication Protocol (TEAP) [22] is an EAP method that allows provisioning a certificate to the peer over a secure encrypted channel established through mutual authentication. EAP-TEAP operates in two phases:

Phase 1: The peer authenticates the server and establishes a secure encrypted tunnel to it with the TLS handshake.

Phase 2: Additional information is exchanged in the encrypted tunnel. This can include execution of another EAP method for peer authentication and provisioning of new credentials to the peer.

Thus, EAP-TEAP allows another EAP-method to be executed in phase 2 so that the inner EAP messages are wrapped inside EAP-TEAP messages and protected by the TLS tunnel. Multiple inner protocols can be executed sequentially in phase 2. Like other TLS-based methods, EAP-TEAP implements message fragmentation.

Two kinds of credentials can be provisioned: a certificate to be used as a long-term credential by the peer, and a Protected Access Credential (PAC), which contains a session ticket and a shared key for fast re-authentication. In the certificate provisioning, the peer sends a PKCS #10 Certification Request to the server and receives the certificate in return. In the optional Server Unauthenticated Provisioning Mode, the peer is unable to validate the server identity in phase 1, but mutual authentication in phase 2 confirms the identity. A limitation of EAP-TEAP is that the outer method is fixed and must use TLS. Also, EAP-TEAP only provisions credentials and cannot provide other configuration data to the peer.

2.4 EAP-CREDS

Credentials Provisioning and Management (CREDS) [13, 14] is a draft specification for secure provisioning and configuration of the peer through EAP; i.e., it shares the goals with the current paper. A single authentication server may perform the authentication, provisioning, and configuration within the EAP protocol.

EAP-CREDS can operate within different outer EAP methods, such as EAP-TLS or EAP-TEAP. The EAP-CREDS specification focuses on the communication of credentials after the outer method has established a secure mutually-authenticated channel. The endpoints can negotiate the credential management protocol, and they can validate the credentials after they have been provisioned. Various credential types can be provisioned including an X.509 certificate, public key, symmetric key, username and password, or one-time password. EAP-CREDS relies on the fragmentation mechanism of the outer EAP method for sending longer messages.

The protocol stacks of EAP-TEAP and EAP-CREDS are illustrated in Figure 2. Note especially how the former implements the outer method while the latter implements the inner method. The puzzling difference between these design choices was one of the observations that motivated the current paper.

3 REQUIREMENTS

As outlined in Section 1, we plan to transfer of short bootstrapping data within the EAP session after the initial authentication and use this data to enable secure credential provisioning and device configuration over HTTPS. We considered making changes to the base EAP protocol, making changes to existing methods, and defining a new tunneled method.

Our system design was driven by the following requirements:

- **R1.** After initial authentication between the peer and the server, the solution should transfer short bootstrapping information over EAP. The bootstrapping information should be sufficient to enable the use of common web technologies for the provisioning of long-term credentials and configuration of the device.
- **R2.** The bootstrapping information is sensitive and should be protected for confidentiality, integrity, and authenticity.
- **R3.** The initial authentication should not be restricted to a single EAP method. It should be possible to use a wide range of EAP methods, i.e., various authentication mechanisms and types of credentials, for the initial authentication.
- **R4.** As always in EAP, the solution should not require changes to the WiFi access point, i.e., the EAP authenticator.
- **R5.** The changes to the base EAP framework should be small or none.
- **R6.** The changes to the existing EAP methods and their implementations should be minimized to make adoption of the solution easier.
- **R7.** The solution must ensure backward compatibility. A peer and server not updated to support the new design must be able to interoperate with the updated end-points — naturally without the new provisioning and configuration functionality.
- **R8.** The additional payloads sent over EAP should be small enough to not require message fragmentation.

4 SYSTEM DESIGN

We first present an example usage scenario in section 4.1 and then describe the generic system architecture and assumptions in sections 4.2–4.4. The bootstrapping process is outlined in section 4.5. The bootstrapping data and different ways of transferring it in the EAP session are described in sections 4.6–4.8.

4.1 Example usage scenario

Alice, who works in the IT support of a university, needs to install a batch of new IoT devices to the campus network. First, she gets the device identifier and the symmetric key of each device by scanning a QR code that shipped with the device. She saves this information to the IT asset-management database. Alice then installs the devices in their intended locations on the campus and powers them on. The rest happens automatically. Within a couple of minutes, all the new IoT devices have been provisioned with long-term credentials and configured to connect to the correct online servers. Each device reboots and joins the campus WiFi network with its long-term

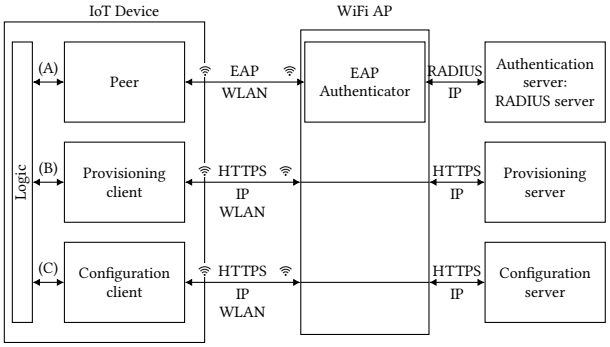

Figure 3: System model.

credentials. The bootstrapping credential in the QR code is no longer valid.

4.2 System architecture

The system architecture is shown in Figure 3.

Active entities: The new device trying to connect to the network comprises the *EAP peer*, *provisioning client*, and *configuration client*. The *authentication server* verifies the initial credentials of the EAP peer and decides whether to grant network access. The *provisioning server* provides a certificate or other long-term credentials to the provisioning client upon valid request. The *configuration server* provides additional data to the configuration client, such as application-layer parameters or a device software update. Both the provisioning and configuration server verify the access rights of the client upon connection.

The WiFi access point (AP) provides and controls physical access to the network. Its *EAP authenticator* component conveys EAP messages between the peer and the authentication server by tunneling them over the RADIUS protocol. It also executes the access decisions made by the authentication server.

Restricting network access: The access point always blocks network access by the peer until it receives a RADIUS Access-Accept message from the authentication server. This message contains the session key MSK created in the EAP method, i.e., the authenticated key exchange. The message both instructs the access point to allow the IoT device to connect to the wired network and gives it the session key for encrypting the wireless connection.

Depending on the capabilities of the access point, it may restrict the client's network access by isolating it to a specific virtual network (VLAN) or by filtering its connections. These restrictions are not decided by the access points itself but, rather, they are based on instructions it receives from the authentication server over the RADIUS protocol. Not all access points support such per-station access control for the wireless clients. However, most access points support multiple wireless network names (SSID) and can restrict access based on the SSID to which the clients connect. Only inexpensive home wireless routers with integrated access point may not have even this capability.

The bootstrapping process depends on a couple of assumptions that need to be explained next.

4.3 Network discovery

The device needs to know to connect to the correct wireless network. The problem is that an off-the-shelf IoT device does not know the name of its new owner, organization, or their networks. There are several different approaches to the network selection. If the device has a sufficient user interface, the simplest approach is to scan the WiFi channels and let the user select the SSID from a list. This will not work for small IoT devices that sometimes only have a static bar code for configuration, though.

Another approach is for the networks to advertise their support of the bootstrapping process. While the 802.11 beacon frames have a mechanism for advertising the network capabilities, using it would require a lengthy standardization process and is not a realistic path forward. Instead, we can standardize an SSID, such as "*provisioning_and_configuration*", for the initial connection. The AP does not need to broadcast this SSID; instead, it should respond to probe messages from wireless clients. Note that there may be multiple access points within the wireless range that respond to the probe, and they may not all belong to the correct organization. Thus, the new device may need to try to connect to several access points before it finds one where the authentication succeeds.

Several different network discovery mechanisms can be deployed in parallel. We believe the user selection approach and the hidden well-known SSID approach are two practical ones that are sufficient to cover most usage scenarios.

4.4 Initial authentication credential

We assume that the new IoT device has some way of performing an initial mutual authentication with the authentication server. This can be based on a preconfigured secret, user-assisted out-of-band channel [4, 19], or any other method available. Indeed, it was one of our design requirements (R3) that we should not limit the initial authentication methods that may be used.

In the example usage scenario, the initial authentication is done with a preconfigured secret (random number of at least 16 octets), which the manufacturer has stored in the device and printed on the QR code that is included in the packaging. Since this secret could leak, it should only be used for the initial authentication during bootstrapping and discarded once the device has been configured and successfully tested network access with the newly provisioned long-term credentials. (A new recovery secret can be provisioned in the device configuration stage to prepare for failure recovery and device reuse after hard reset.)

Since the secret in the QR code could leak from or be tampered within the supply chain, this mechanism may not be suitable for high-security systems. For most applications, the risk is relatively low because the same attacker needs to be physically within the wireless range of the device or access point on the right day to spoof the other in the bootstrapping process.

4.5 Bootstrapping process

The bootstrapping process includes the following three stages, which correspond directly to the client and server components in Figure 3.

Initial authentication. When the new IoT device is powered on, it uses one of the available discovery mechanisms to select

the WiFi network. In our example scenario, the device probes for the well-known SSID. The device then tries to connect to the discovered network(s). The EAP peer on the device tries to perform initial authentication with an EAP method that is chosen by the authentication server and supported by the peer. After successful authentication, still during the EAP session, the server transfers bootstrapping data to the IoT device. (The contents of bootstrapping data will be described in section 4.6 and the methods for transferring it in section 4.8.)

After the success successful initial authentication, the EAP authenticator also grants the device access to the wired IP network so that it can connect to the provisioning and configuration servers. Ideally, the access would be restricted to only these servers. This restriction can be implemented either with advanced features in the access point or by isolating all access through the well-known bootstrapping SSID to a virtual or physical network that has no other connectivity.

The device then connects to the provisioning and configuration servers over secure web protocols, i.e., HTTPS APIs. The device authenticates the servers based on certificate hashes received in the bootstrapping data. To authenticate itself to the servers, the device attaches a token received in the bootstrapping data to the HTTP request. It sends the token in the *Authorization: Bearer* header.

Provisioning. In the credential provisioning, the device sends a certificate signing request to the provisioning server, which issues an X.509 certificate to the device. The certificate serves as its long-term credential for network access and possibly for other services. This stage establishes the identity of the device in the network. Other types of credentials could be issued in addition to or instead of the certificate.

Configuration. The minimum configuration data which the client must retrieve from the configuration server includes the SSID, the root CA certificate, and the authentication server's domain name. With these, the device can identify and authenticate the correct wireless network for future network access. The device may access multiple HTTP APIs at the configuration server. The API endpoints for the most common configuration information should be standardized, but there may also be need for manufacturer- or industry-specific configuration APIs. The configuration stage may also include downloading of software updates which must be installed before connecting to the actual access network. After receiving and processing the configuration data, the device reconnects to the access network SSID and authenticates itself with the newly-acquired certificate and network information.

Why are the provisioning and configuration servers separate? In fact, these services are just API endpoints identified by URLs and server certificate hashes, and the two may reside on same server. We allow the separation because many organizations want to isolate certificate provisioning to a separate high-security server that has no other functions.

4.6 Contents of bootstrapping data

As already mentioned, the bootstrapping data includes the API endpoint addresses (HTTPS URLs) of the provisioning and configuration servers. There can be either domain names or IP addresses in the URLs. The bootstrapping data also includes the hashes of the two server certificates. The reason is that the new device does not know the owner's preferred root CA before completing the configuration stage. We do not want to rely on the common web PKI; while it may be acceptable for some users and applications, it is not for all.

The client token is a JSON Web Token (JWT) [10]. It has three fields: (1) the header, which indicates the purpose of the access, e.g., "provisioning", or "configuration"; (2) the payload, which contains the long-term identity of the device in the network and an expiration time that limits the validity time of the token to, e.g., five minutes; and (3) the authentication server's digital signature over the previous fields. The clocks in provisioning and configuration servers have to be roughly synchronized with the time in the authentication server, so that they can check the validity time of the client token. The signature algorithm in our implementation is ES256; it is an Elliptic Curve Digital Signature Algorithm (ECDSA) using the P-256 elliptic curve and SHA-256 cryptographic hash function [9].

The total size of bootstrapping data in our implementation is about 800 bytes. Therefore, the bootstrapping data structure fulfills the requirement R7 about avoiding fragmentation.

4.7 Security of the bootstrapping data

The short bootstrapping data needs to be protected for integrity and authenticity as well as for confidentiality. This is the task of the EAP method that transfers the data to the peer. There are two ways to achieve this protection. The first is to use a TLS-based outer method and transfer the data inside the TLS tunnel. The second is to use authenticated encryption and a key derived from the session key MSK which was created in the same EAP method. In the latter case, the transfer of the bootstrapping data needs to take place at the end of the EAP session when the MSK is available.

The confidentiality protection for the bootstrapping data is needed because the included JWT token is a critical secret. The possession of the signed token is what authorizes the peer to access the provisioning server and to obtain a secure long-term identity. The limited lifetime of the token does, however, reduce the risk of the token leaking.

The integrity and authenticity protection for the bootstrapping data is needed to ensure that the peer receives the correct URLs and certificate hashes. These ensure that the peer connects to the correct HTTPS servers and API endpoints after the initial authentication. Moreover, the JWT token needs the same integrity and authenticity protection during its transfer from the authentication server to the peer. This is because swapping tokens could confuse the provisioning and configuration servers (and the peer itself) about the peer identity. Note that the token signature is not checked by the peer.

4.8 Transferring bootstrapping data in EAP

We have identified four architecturally different ways of transferring the bootstrapping data in an EAP session: (1) in an EAP notification message; (2) as an additional attribute in an existing EAP method-specific message; (3) in a new inner EAP method; and (4) using a combination of an outer and an inner EAP method. We implemented all four methods to ensure that they are feasible.

Table 1: Comparison of the four methods for transferring bootstrapping data

Requirement from section 3	1. Notification message	2. Method attribute	3. Inner EAP method	4. Outer and inner EAP method
R1 (bootstrapping data)	Yes	Yes	Yes	Yes
R2 (protect the data)	Yes	Yes	Yes	Yes
R3 (any initial method)	Yes	No	No	Yes
R4 (unmodified AP)	Yes	Yes	Yes	Yes
R5 (unmodified EAP)	No	Yes	Yes	Yes
R6 (unmodified methods)	Yes	Partially	Yes	Yes
R7 (backward compatibility)	Partially	Yes	Yes	Yes
R8 (no fragmentation)	Yes	Yes	Yes	Yes

Figure 1 summarizes how these solutions fulfill the requirements of section 3. All the solutions achieve the basic goals (R1), there is no impact on the WiFi access point (R4), and the bootstrapping data is smaller than one kilobyte (R8). The other requirements will be discussed below. We obviously limit the discussion to solutions that protect the bootstrapping data (R2).

4.8.1 EAP notification message. The EAP framework defines a notification message type for conveying a human-readable message to the peer. The default maximum length of a notification message is 1020 bytes. Since this message type is part of the EAP framework itself, it is available in all EAP methods [21]. It could be used to transfer the bootstrapping data to the peer just before the final success message in the EAP session.

R3 and R5 are fulfilled because the mechanism is independent of the EAP method. A legacy EAP peer that receives a notification message with bootstrapping data will be able to successfully complete the EAP session with the authentication server. However, it may try to display the bootstrapping data, which could cause some confusion for the end-user. Thus, R7 is only partially fulfilled.

Since the notification message is not secured by default, the bootstrapping data should be protected cryptographically. We have worked out how to do this with the session key produced by the authentication in the EAP method and implemented the protection (see section 5.2), thus fulfilling R2. The trade-off is that this is a change to the base EAP specification and, thus, R5 is not met.

Some issues arise with using notification message for the bootstrapping data. First, the server does not receive any meaningful feedback from the peer. The peer acknowledges the notification with an empty response [21]. Second, since EAP is widely deployed and all existing EAP methods rely on the EAP framework specifications [15, 21], even small changes in the EAP framework may be difficult to standardize.

4.8.2 New attributes in method-specific messages. Several EAP methods send optional attributes in their EAP message payloads. For example, EAP-TTLS uses Attribute-Value Pair (AVP) objects after the completion of the TLS handshake, while EAP-PEAP, EAP-TEAP, EAP-FAST, EAP-POTP use a Type-Length-Value (TLV) format. A new attribute could be added to convey the bootstrapping data.

R3 and R6 are not fulfilled because this method is specific to the EAP method and requires at least small changes to its implementation. In addition, when sending the attribute in the middle of the EAP session, we do not have access to the session key. For that reason, we cannot provide a separate mechanism for securing the attribute and need to depend on security provided by the underlying EAP method. This limits the set of EAP methods where this solution can be used without compromising R2.

No changes to the base EAP are necessary (R5), and a legacy client will simply ignore an optional attribute containing bootstrapping data (R7).

4.8.3 New inner EAP method. Several existing EAP methods set up a TLS tunnel for communication (e.g., EAP-TTLS, EAP-PEAP, EAP-TEAP and EAP-FAST), and it is possible to execute another, inner EAP method within that TLS tunnel. We created a new inner EAP method, called EAP-iPROV (EAP-Inside PROVisioning), for sending the bootstrapping data. The protocol stack with EAP-iPROV is shown in Figure 4.

The three pairs of messages exchanged in the inner method EAP-iPROV are shown in Figure 5. The first is a standard EAP Identity request. The server then requests the peer to run the EAP-iPROV method, and the peer agrees. The transfer of the bootstrapping data takes place in the last request, which the peer acknowledges.

The use of EAP-iPROV is negotiated between the server and peer, and the bootstrapping data is sent to the peer only if the peer indicates that it is able to receive it. More specifically, when the authentication server initiates the inner EAP method (EAP-iPROV), a legacy peer will reply with a NAK message. The peer can then complete outer EAP method authentication. Thus, requirement R7 is fulfilled.

Requirement R2 is fulfilled assuming that the EAP-iPROV messages are sent inside a secure tunnel created by the outer EAP method. This assumption is true for the TLS-based EAP methods. The requirement R3 is not fulfilled, since only a TLS-based subset of existing EAP methods have the mechanism to invoke an inner EAP method. We also say that R8 is fulfilled because the TLS-based methods implement fragmentation and EAP-iPROV does not need to consider it.

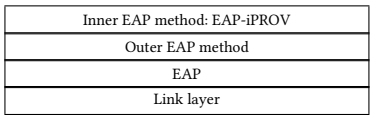

Figure 4: EAP-iPROV protocol stack

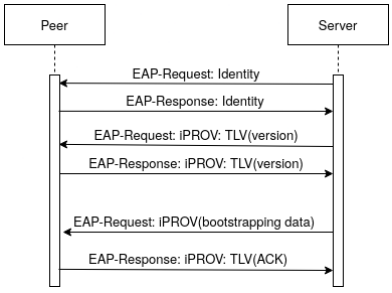

Figure 5: Messages in EAP-iPROV when the peer wants the configuration data

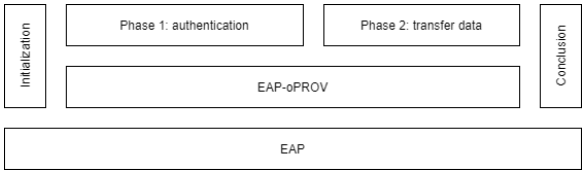

Figure 6: EAP-oPROV session timeline

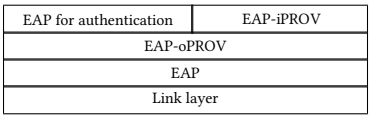

Figure 7: EAP-oPROV protocol stack

Requirements R5 and R6 are fulfilled because there is no need to modify the base EAP or any existing EAP method. Of course, the inner EAP method, EAP-iPROV, needs to be standardized and implemented, and the outer method needs to be configured to use this new inner method.

4.8.4 Combination of outer and inner EAP method. This method, called EAP-oPROV (EAP Outside PROVisioning), is an outer EAP method that first (phase 1) invokes another EAP method for the actual authentication and then (phase 2) uses the created session keys to protect the bootstrapping data. For modular implementation of the latter step, it invokes the EAP-iPROV method describe above. The EAP-oPROV session timeline is illustrated in Figure 6, and the matching EAP-oPROV protocol stack is shown in Figure 7.

One advantage of this solution is that it supports most EAP methods for the initial authentication and requires no changes to them or to the base EAP protocols (requirements R3, R5 and R6).

The backward compatibility requirement R7 is met because, when the authentication server initiates EAP-oPROV method, a legacy peer will reply with a NAK message; and the server will then be able to initiate another EAP method. Fragmentation is avoided (R8) if the bootstrapping data is limited to about one kilobyte.

Based on the comparison in Table 1, this solution meets the requirements best.

5 IMPLEMENTATION

This section describes our implementation of the bootstrapping process.

5.1 Authentication server and WiFi client implementation

The EAP authentication server is implemented by extending *hostapd*. The EAP peer, i.e., WiFi client, extends *wpa_supplicant*. Both are open-source implementations by Jouni Malinen [11]. Our implementation adds the EAP-oPROV and EAP-iPROV methods. The source code can be found in [1].

Depending on the solution version, the server and client are configured to support one or both of the new methods and to also invoke EAP-TTLS as the TLS-based method when needed. Recall that several EAP methods may be called in a single session one after the other. This requires adding configuration parameters in order to describe the more complex session.

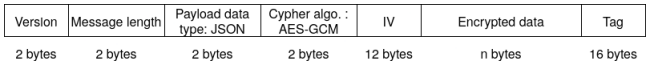

Figure 8: Encrypted bootstrapping data and AAD in phase 2 of EAP-oPROV

5.2 Protection of bootstrapping data

Section 4.7 concluded that the authenticity and confidentiality of client tokens must be protected by the EAP method. EAP-oPROV protects the token confidentiality by deriving a symmetric key from the Master Secret Key (MSK), which is produced by the successful conclusion of the phase 1 EAP method, and using the derived symmetric key to encrypt the EAP-iPROV messages in phase 2 of EAP-oPROV.

The symmetric key K is derived from MSK as follows:

$$K = \text{hmac\_sha256\_kdf(MSK, label, salt, length)},$$

where label is a string consisting of the concatenation of "Derive EAP-iPROV message key", the name of EAP method used in phase 1, and the identifier of EAP-oPROV; salt is the peer identifier; and the requested output length is 16 bytes.

The derived symmetric key K is used in authenticated encryption of the messages with the AES-GCM algorithm that outputs the encrypted payload *enc* and the verification tag *tag*:

$$(enc, tag) = \text{aes\_gcm(message, AAD, } K\text{, IV)}.$$

The additional associated data (AAD) includes information on the encryption algorithm, version, and payload data type; and the IV parameter consists of 12 random bytes.

The encrypted message in EAP-oPROV phase 2 contains the elements AAD and IV, the encrypted payload, and the verification tag. Its structure is shown in Figure 8. The total size of the protected bootstrapping data in our implementation is about 800 bytes.

5.3 Provisioning and configuration servers and clients

The EAP peer conveys the received client tokens to the provisioning and configuration clients in the same machine. We implemented both clients as Python scripts. The clients uses the tokens to access the provisioning and configuration servers over HTTPS.

The provisioning of the certificate to the IoT device is done with the Enrollment over Secure Transport (EST) protocol [16]. We used *Simple Python EST server + CA* [18] as the provisioning server. We implemented the configuration server as a simple web API server in Python. Both the provisioning and configuration servers have REST APIs and they are accessed over HTTPS.

5.4 Implementation complexity

Table 2 summarizes the coding work for the combined EAP-oPROV and EAP-iPROV solution. In total, we added about 4000 lines of C and 1000 lines of Python to existing open-source software. About half of the new C code (~ 2000 lines) is in the wpa_supplicant (EAP

peer), and the other half is in hostapd (authentication server) [11]. The bulk of the new Python code is in the provisioning client written by us. Less than 10 lines of Python were added in the provisioning server [18] to validate the client tokens.

All in all, the coding effort needed to implement our approach is small. This is partly due to our ability to reuse the existing EAP framework and methods to compose the new solution. Even more importantly, our early design choices meant that we could implement the provisioning and configuration clients and the configuration server in a high-level language and modern web software frameworks, which help to minimize the amount of code.

6 EXPERIMENTS

This section reports on the experiments made with our implementation of the two solutions that use HTTPS the provisioning and configuration stages: EAP-iPROV and combined EAP-oPROV and EAP-iPROV. We also compare them to the fully EAP-based EAP-TEAP, which was modified to convey some configuration data. We are unable to compare with EAP-CREDS, because the specification is still work in progress and there are no implementations available.

6.1 Experimental setup

Our experimental setup and software components are represented in Figure 9. The IoT device is Raspberry Pi 4 model B. The WiFi access point (AP) and all three servers run on Raspberry Pi 3 model B+. The servers have the same IP address but different port numbers. The two devices communicate with each other over WiFi. For development and testing, both devices were also connected to a router with Ethernet cables, so that the developer can connect to them with ssh. The file systems of both Raspberry Pi devices were also mounted to the developer's computer.

During the experiments, we recorded messages exchanged on the WiFi interface of the server Raspberry Pi with tcpdump, and

Table 2: Implementation sizes

Entity	Total code	New code		
Hostapd authentication server and authenticator	751×10^3 lines (22 MB) C code	C code	EAP-oPROV	1127 lines (33 kB)
			EAP-iPROV	791 lines (20 kB)
		Python code	Token generation	118 lines (4 kB)
Wpa_supplicant peer	747×10^3 lines (21.9 MB) C code	C code	EAP-oPROV	1025 lines (30 kB)
			EAP-iPROV	797 lines (21 kB)
Provisioning client (custom made)	600 lines (19 kB) Python code			
Provisioning server [18]	209 lines (8 kB) Python code, 86 lines (6 kB) Bash code	Python code	Token verification	5 lines (< 1 kB)
Configuration client (custom made)	70 lines (3 kB) Python code			
Configuration server (custom made)	79 lines (3 kB) Python code	Python code	Token verification	5 lines (< 1 kB)

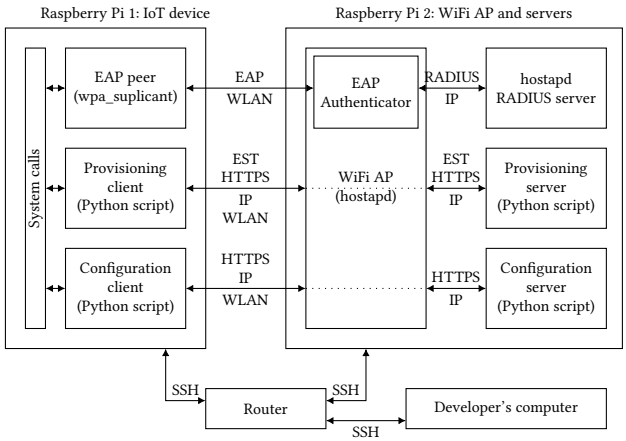

Figure 9: Development environment and experiment setup

then parsed these traces in the developer's computer with Wireshark. This simple method allowed full view of the communication between the servers and the peer.

6.2 Experiment results

We conducted a series of experiments in order to compare our approach with the case were the authentication, provisioning, and configuration are all done in EAP-TEAP in the same EAP session.

As explained in section 2.3, EAP-TEAP can send a certificate to the peer over a TLS tunnel that is established by the method. There is no mechanism for sending configuration data, but it is possible to send a Vendor-Specific TLV over the TLS tunnel, and we implemented this extension to EAP-TEAP [11]. Naturally, both the peer and the authentication server have to understand how to interpret this non-standard TLV.

We conducted a series of experiments to evaluate the latency and communication overhead of different protocol options. Configuration data sizes can range from a few bytes to a full firmware update. First, we varied the size of the configuration data. In the tested implementation of EAP-TEAP, the EAP peer exited with failure when the configuration data was about 16 kB or larger. The reason is that the hostapd implementation of EAP-TEAP limits the size of any message to 16 kB. The probable reason for this limitation is that the maximum record size in TLS is 2^{14} bytes [17]. Our solution variants that retrieve the configuration over a web API and not in the EAP session have not such limitations.

Second, we fixed the sizes of the provisioned certificate and configuration data to about one kilobyte and 400 bytes, respectively. Such configuration data could include, for example, network parameters.

i. As a baseline, we measured the performance of EAP-TEAP with and without provisioning and configuration. A shell script invoked EAP-TEAP repeatedly for 20 minutes, and *tcpdump* was used to collect the network traces. EAP-TEAP was configured so that both client and server use certificates for the TLS handshake. We ran EAP-TEAP both with and without provisioning and configuration. The results of this experiment are summarized in rows (0) and (1) of Table 3.

Table 3: Comparison of provisioning and configuration with EAP-TEAP, EAP-oPROV, and EAP-iPROV with EAP-TTLS

Protocol	Messages from server	Bytes from server	Messages from client	Bytes from client	Duration (seconds)
0. EAP-TEAP without provisioning or configuration	10	1862	9	2196	0.42 [0.302, 0.659]
1. EAP-TEAP with provisioning and configuration	10	3092	9	2989	0.4635 [0.367, 0.621]
2. EAP-TTLS / EAP-iPROV	10 [10, 11]	2400 [2392, 3177]	9 [9, 10]	2094 [2094, 2167]	0.943 [0.856, 1.094]
3. EAP-oPROV / EAP-TTLS + EAP-iPROV	11 [11, 12]	2432 [2416, 3179]	10 [10, 11]	2150 [2150, 2193]	0.981 [0.879, 1.148]
4. HTTPS provisioning	15	9594	9	3257 [3256, 3257]	0.23 [0.22, 0.34]
5. HTTPS configuration	7	3875 [3874, 3876]	5	1759	0.03 [0.029, 0.031]
6. HTTPS provisioning and configuration	22 [15, 23]	13468 [10260, 14918]	14 [10, 14]	5016 [3343, 5016]	0.88 [0.83, 1.18]
7. EAP-TTLS/ EAP-iPROV + HTTPS provisioning and configuration	32 [25, 34]	15868 [12452, 18095]	23 [19, 24]	7110 [5337, 7183]	1.823 [1.686, 2.274]
8. EAP-oPROV / EAP-TTLS + EAP-iPROV + HTTPS prov and conf	33 [26, 35]	15900 [14676, 18097]	24 [20, 25]	7166 [5493, 9376]	1.69 [1.709, 2.328]

ii. Next, a similar script repeatedly invoked EAP-oPROV with inner methods EAP-TTLS (for mutual authentication) and EAP-iPROV (for sending provisioning and configuration tokens), followed by provisioning and configuration over HTTPS. EAP-TTLS was configured so that both the client and server have certificates for the TLS handshake. The results of this experiment are summarized in rows (3) and (8) of Table 3.

iii. The measurements were repeated with EAP-TTLS as the outer method and EAP-iPROV as the inner method, followed by provisioning and configuration over HTTPS. The results of this experiment are summarized in rows (2) and (7) of Table 3. About 100 runs were done for each protocol option during the experiment.

Table 3 shows the number of messages and bytes sent in each protocol variant. When there was variation in the results, we show the median value as well as the observed value range in brackets. To summarize, rows 0-4 show results for protocols that communicate only in the EAP session, rows 4-6 show only the HTTPS API access, and rows 7-8 show solutions that only send short bootstrapping data over EAP and then continue with HTTPS. Note that the HTTPS communication has not been optimized: the client creates a new TLS session with the server for each EST protocol request.

Looking at Table 3 we can observe the following:

– Comparing protocol (0) and (1) we see that adding provisioning and configuration to EAP-TEAP authentication adds about 2 kB, and less than 0.1 s.

– Protocol (1) takes roughly 0.5 s, while protocols (2) and (3) take about one second. The reason for the additional time in the latter is the initialization of the two EAP methods: EAP-TTLS and EAP-iPROV.

– The number of messages in protocols (1), (2), and (3) is about 10. Note that (3) has one message and 32 bytes more than (2), and it takes 0.04 second more time. This is because there is one extra EAP Hello message in EAP-oPROV.

– Less data is exchanged in protocol (2) or (3) than in (1) because in (2) and (3) the actual provisioning and configuration happen outside of the EAP exchange.

– Protocol (4) has more messages and takes longer than (5) because the EST protocol used in (4) includes several round trips, while (5) is completed within a single round trip.

– The time for running (4) and (5) is about 0.3 second; but (6) takes about 0.9 second. The reason is that in (6), the operations (4) and (5) are run inside a Python script in the IoT device, which adds the overheads of context switching and compilation.

– The transfer of configuration and provisioning data over HTTP adds about 14 kB. This overhead could be reduced. For example, since each TLS handshake involves exchange of about two kilobytes, reducing the number of TLS handshakes from three to one in EST provisioning, would reduce the amount data by about four kilobytes. We leave these optimizations to future work.

All in all, in our experiments, the provisioning and configuration with EAP-TEAP is faster (by about 0.5 s) than EAP-oPROV/EAP-iPROV. The amount of data exchanged inside EAP-TEAP is larger (by about 600 B) than that inside EAP-oPROV/EAP-iPROV. The reason is that, in the latter protocols, the actual provisioning and configuration take place over HTTPS after EAP-oPROV success.

We argue, however, that the performance differences are not great enough to affect the usability of a provisioning and configuration process that takes place automatically and only once for each device. On the other hand, the observed limit in the size the configuration data in the fully EAP-based solution is indicative of the difficulty of implementing generic and flexible solutions as binary protocols inside the protocol stack, in comparison to using web technologies whenever possible.

7 DISCUSSION

We have shown that the idea of transferring small bootstrapping data inside EAP and then using HTTPS for most work is a flexible and easily implementable design strategy. We also explained the need for protecting the authenticity and confidentiality of the data transfer.

Attacker model: Considering the attacker model on a higher level, we assume that an attacker has full control of the radio channels between IoT device and WiFi access point. Most importantly, the attacker is within the wireless range, it can connect to the WiFi access point and establish EAP sessions with authentication server, and it can set up a fake access point with the same SSID as the authentic one.

Since IoT devices come from various manufacturers and levels of quality, they cannot be fully trusted. Trusting a device for a specific application is a user decision and outside the scope of the current paper. However, a rogue device should not be able to compromise the security of other devices. That is why the bootstrapping process carefully protects the device identity information from the initial authentication via the token to the certificate provisioning. The attacker most not be able to create counterfeit tokens, capture the token of an honest device, or to cause the honest device to present the wrong token.

The one remaining threat that we cannot solve is that an attacker or a rogue user in control of multiple devices can swap and transfer the identities between these devices — either by swapping the tokens during the bootstrapping or by leaking the devices' private keys after certificate provisioning. A potential countermeasure

against these residual threats is an attestable hardware-based device identity. For example, the Keystore Attestation feature in Android devices assures that the private key has been stored inside secure device hardware [3].

Future extensions: Most extensions to our solution should be implemented as extensions to the configuration stage by defining new configuration API endpoints. The outer method EAP-oPROV presented in this paper does, however, provide a generic mechanism for extending the functionality of EAP. The primitives it provides can be used to add further secure messages between the peer and authentication server to the EAP session.

8 CONCLUSION

EAP is widely used for peer authentication in wireless networks. On the other hand, its use for client management is limited to one method, EAP-TEAP [22], which can send a certificate to the client. A draft proposal EAP-CREDS [13] suggests broadening the functionality with more diverse credential provisioning and client configuration features. Both protocols transfer all data inside the EAP session.

We suggest a simpler and more modern approach: send only short bootstrapping data inside the EAP method and implement the provisioning and configuration using web APIs. The bootstrapping data consists of the next server addresses and client tokens for accessing them. We show that this approach can be implemented with very reasonable coding effort and results in a highly flexible solution that allows free choice of the initial authentication method, can make use of existing standard credential provisioning protocols, and is easy to extend by creating new web APIs.

We compared four architecturally different protocol designs for sending the bootstrapping data: (1) EAP notification message, (2) using method-specific extensible attributes that are available on some methods, (3) a new EAP method EAP-iPROV that is invoked inside a secure tunnel created by an outer TLS-based method and only transfers the bootstrapping data, and (4) a new outer EAP method EAP-oPROV that wraps first the actual authentication method and then EAP-iPROV for transferring the bootstrapping data. These designs were implemented by extending the open-source hostapd and wpa_supplicant software components and evaluated by building an experimental testbed on Raspberry Pi devices. We provide a thorough comparison of the solutions, with (4) coming out on top. We also compare with EAP-TEAP, which has somewhat better raw performance but loses on flexibility and ease of future development.

REFERENCES

[1] Tolgahan Akgun and Sebastien Boire. 2021. *EAP-OPROV.* https://github.com/Sebastien2/EAP-PROV

[2] Open Mobile Alliance. 2017–2021. *OMA Specifications.* Open Mobile Alliance. http://openmobilealliance.org/wp/index.html

[3] Android documentation. 2020. *Verifying hardware-backed key pairs with key attestation.* https://developer.android.com/training/articles/security-key-attestation

[4] Tuomas Aura, Mohit Sethi, and Aleksi Peltonen. 2021. *Nimble out-of-band authentication for EAP (EAP-NOOB).* Internet-Draft draft-ietf-emu-eap-noob-04. Internet Engineering Task Force. https://www.ietf.org/archive/id/draft-ietf-emu-eap-noob-04.txt

[5] Cisco. 2021. *Implementing Network Admission Control Phase One Configuration and Deployment.* https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Security/ImplNAC/ImpNAC/nac01.html

[6] Geoffrey Cooper, Brad Behm, Ankur Chakraborty, Hanu Kommalapati, Giri Mandyam, and Hannes Tschofenig. 2020. *FIDO Device Onboard Specification.* Review Draft December 02, 2020. FIDO Alliance. https://fidoalliance.org/specs/FDO/FIDO-Device-Onboard-RD-v1.0-20201202.html

[7] Open Connectivity Foundation. 2021. *OCF Specification 2.2.3, April 14, 2021.* https://openconnectivity.org/developer/specifications/

[8] IBM. 2021. *IBM Watson IoT Platform.* https://internetofthings.ibmcloud.com/

[9] Michael Jones. 2015. JSON Web Algorithms (JWA). RFC 7518. https://doi.org/10.17487/RFC7518

[10] Michael Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Token (JWT). RFC 7519. https://doi.org/10.17487/RFC7519

[11] Jouni Malinen. 2013. *hostapd and wpa supplicant.* https://w1.fi/

[12] Microsoft. 2021. *NAP Server-side Architecture.* https://docs.microsoft.com/en-us/windows/win32/nap/nap-server-side-architecture

[13] Massimiliano Pala. 2020. *Credentials Provisioning and Management via EAP (EAP-CREDS).* Internet-Draft draft-pala-eap-creds-07. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-pala-eap-creds-07 Work in Progress.

[14] Massimiliano Pala. 2020. *EAP-CREDS: Enabling Policy-Oriented Credential Management in Access Networks.* https://www.cablelabs.com/eap-creds-enabling-policy-oriented-credential-management-in-access-networks

[15] Nick L. Petroni, John Vollbrecht, Yoshihiro Ohba, and Pasi Eronen. 2005. State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator. RFC 4137. https://doi.org/10.17487/RFC4137

[16] Max Pritikin, Peter E. Yee, and Dan Harkins. 2013. Enrollment over Secure Transport. RFC 7030. https://doi.org/10.17487/RFC7030

[17] Eric Rescorla and Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. https://doi.org/10.17487/RFC5246

[18] Jukka-Pekka Sarjanen. 2019. *Simplest Git repository for EST provisioning.* https://github.com/abrox/simplest

[19] Mohit Sethi, Elena Oat, Mario Di Francesco, and Tuomas Aura. 2014. Secure Bootstrapping of Cloud-Managed Ubiquitous Displays. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '14).* ACM, 739–750. https://doi.org/10.1145/2632048.2632049

[20] John Vollbrecht and Larry Blunk. 1998. PPP Extensible Authentication Protocol (EAP). RFC 2284. https://doi.org/10.17487/RFC2284

[21] John Vollbrecht, James D. Carlson, Larry Blunk, Dr. Bernard D. Aboba, and Henrik Levkowetz. 2004. Extensible Authentication Protocol (EAP). RFC 3748. https://doi.org/10.17487/RFC3748

[22] Hao Zhou, Nancy Cam-Winget, Joseph A. Salowey, and Steve Hanna. 2014. Tunnel Extensible Authentication Protocol (TEAP) Version 1. RFC 7170. https://doi.org/10.17487/RFC7170