Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 14, 2019

W. Kumari
Google
C. Doyle
Juniper Networks

June 12, 2019

Secure Device Install
draft-wkumari-opsawg-sdi-04

Abstract

Deploying a new network device often requires that an employee physically travel to a datacenter to perform the initial install and configuration, even in shared datacenters with "smart-hands" type support. In many cases, this could be avoided if there were a standard, secure way to initially provision the devices.

This document extends existing auto-install / Zero-Touch Provisioning mechanisms to make the process more secure.

[Ed note: Text inside square brackets ([]) is additional background information, answers to frequently asked questions, general musings, etc. They will be removed before publication. This document is being collaborated on in Github at: https://github.com/wkumari/draft-wkumari-opsawg-sdi. The most recent version of the document, open issues, etc should all be available here. The authors (gratefully) accept pull requests.]

[Ed note: This document introduces concepts and serves as the basic for discussion - because of this, it is conversational, and would need to be firmed up before being published]

This Internet-Draft will expire on December 14, 2019.

Copyright Notice

Table of Contents

1. Introduction

In a growing, global network, significant amounts of time and money are spent simply deploying new devices and "forklift" upgrading existing devices. In many cases, these devices are in shared datacenters (for example, Internet Exchange Points (IXP) or "carrier neutral datacenters"), which have staff on hand that can be contracted to perform tasks including physical installs, device reboots, loading initial configurations, etc. There are also a number of (often vendor proprietary) protocols to perform initial device installs and configurations - for example, many network devices will attempt to use DHCP to get an IP address and configuration server, and then fetch and install a configuration when they are first powered on.

Network device configurations contain a significant amount of security related and / or proprietary information (for example, RADIUS or TACACS+ secrets). Exposing these to a third party to load onto a new device (or using an auto-install techniques which fetch an (unencrypted) config file via something like TFTP) is simply not acceptable to many operators, and so they have to send employees to remote locations to perform the initial configuration work. As well as having a significant monetary cost, it also takes significantly longer to install devices and is generally inefficient.

There are some workarounds to this, such as asking the vendor to pre-configure the devices before shipping it; asking the smart-hands to install a terminal server; providing a minimal, unsecured configuration and using that to bootstrap to a complete configuration, etc; but these are often clumsy and have security issues - for example, in the terminal server case, the console port connection could be easily snooped.

This document layers security onto existing auto-install solutions to provide a secure method to initially configure new devices. It is optimized for simplicity, both for the implementor and the operator; it is explicitly not intended to be an "all singing, all dancing" fully featured system for managing installed / deployed devices, nor is it intended to solve all use-cases - rather it is a simple targeted solution to solve a common operational issue. Solutions

such as Secure Zero Touch Provisioning (SZTP)" [RFC8572] are much more fully featured, but also more complex to implement and / or are not widely deployed yet.

1.1. Requirements notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Overview / Example Scenario

Sirius Cybernetics Corp needs another peering router, and so they order another router from Acme Network Widgets, to be drop-shipped to the Point of Presence (POP) / datacenter. Acme begins assembling the new device, and tells Sirius what the new device's serial number will be (SN:17894321). When Acme first installs the firmware on the device and boots it, the device generates a public-private keypair, and Acme publishes it on their keyserver (in a certificate, for ease of use).

While the device is being shipped, Sirius generates the initial device configuration, fetches the certificate from Acme keyservers by providing the serial number of the new device. Sirius then encrypts the device configuration and puts this encrypted config on a (local) TFTP server.

When the device arrives at the POP, it gets installed in Sirius' rack, and cabled as instructed. The new device powers up and discovers that it has not yet been configured. It enters its autoboot state, and begins the DHCP process. Sirius' DHCP server provides it with an IP address and the address of the configuration server. The router uses TFTP to fetch its config file (note that all this is existing functionality). The device attempts to load the config file - if the config file is unparsable, (new functionality) the devies tries to uses its private key to decrypt the file, and, assuming it validates, installs the new configuration.

Only the "correct" device will have the required private key and be able to decrypt and use the config file (See Security Considerations). An attacker would be able to connect to the network and get an IP address. They would also be able to retrieve (encrypted) config files by guessing serial numbers (or perhaps the server would allow directory listing), but without the private keys an attacker will not be able to decrypt the files.

This document uses the serial number of the device as a unique identifier for simplicity; some vendors may not want to implement the

system using the serial number as the identifier for business reasons (a competitor or similar could enumerate the serial numbers and determine how many devices have been manufactured). Implementors are free to choose some other way of generating identifiers (e.g UUID [RFC4122]), but this will likely make it somewhat harder for operators to use (the serial number is usually easy to find on a device, a more complex system is likely harder to track).

[Ed note: This example uses TFTP because that is what many vendors use in their auto-install / ZTP feature. It could easily instead be HTTP, FTP, etc.]

3. Vendor Role / Requirements

This section describes the vendors roles and responsibilities and provides an overview of what the device needs to do.

3.1. Device key generation

During the manufacturing stage, when the device is intially powered on, it will generate a public-private keypair. It will send its unique identifier and the public key to the vendor's Certificate Publication Server to be published. The mechanism used to do this is left undefined. Note that some devices may be contrained, and so may send the raw public key and unique identifier to the certificate publication server, while mode capable devices may generate and send self-signed certifcates.

3.2. Certificate Publication Server

The certificate publication server contains a database of certificates. If newly manufactured devices upload certificates the certificate publication server can simply publish these, if the devices provide raw public keys and unique identfiers the certificate publication server will need to wrap these in a certificate. Note that the certificat publication server MUST only accept certifcates or keys from the vendor's manufacturing facilities.

The customers (e.g Sirius Cybernetics Corp) query this server with the serial number (or other provided unique identifier) of a device, and retrieve the associated certificate. It is expected that operators will receive the unique identifier (serial number) of devices when they purchase them, and will download and store / cache the certificate. This means that there is not a hard requirement on the uptime / reachability of the certificate publication server.

```
                         +------------+
        +------+         |Certificate |
        |Device|         |Publication |
        +------+         |   Server   |
                         +------------+
   +----------------+   +--------------+
   |   +---------+  |   |              |
   |   | Initial |  |   |              |
   |   |  boot?  |  |   |              |
   |   +----+----+  |   |              |
   |        |       |   |              |
   | +------v-----+ |   |              |
   | |  Generate  | |   |              |
   | |Self-signed | |   |              |
   | |Certificate | |   |              |
   | +------------+ |   |              |
   |        |       |   |   +-------+  |
   |        +-------|---|-->|Receive|  |
   |                |   |   +---+---+  |
   |                |   |       |      |
   |                |   |   +---v---+  |
   |                |   |   |Publish|  |
   |                |   |   +-------+  |
   |                |   |              |
   +----------------+   +--------------+
```

Initial certificate generation and publication.

4. Operator Role / Responsibilities

4.1. Administrative

When purchasing a new device, the accounting department will need to get the unique device identifier (likely serial number) of the new device and communicate it to the operations group.

4.2. Technical

The operator will contact the vendor's publication server, and download the certificate (by providing the unique device identifier of the device). The operator SHOULD fetch the certificate using a secure transport (e.g HTTPS). The operator will then encrypt the initial configuration to the key in the certifcate, and place it on their TFTP server. See Appendix B for examples.

```
                         +------------+
      +--------+         |Certificate |
      |Operator|         |Publication |
      +--------+         |   Server   |
                         +------------+
   +----------------+  +----------------+
   | +-----------+  |  |  +-----------+ |
   | |   Fetch   |  |  |  |           | |
   | |  Device   |<------>|Certificate| |
   | |Certificate|  |  |  |           | |
   | +-----+-----+  |  |  +-----------+ |
   |       |        |  |                |
   | +-----v------+ |  |                |
   | |  Encrypt   | |  |                |
   | |   Device   | |  |                |
   | |   Config   | |  |                |
   | +-----+------+ |  |                |
   |       |        |  |                |
   | +-----v------+ |  |                |
   | |  Publish   | |  |                |
   | |    TFTP    | |  |                |
   | |   Server   | |  |                |
   | +------------+ |  |                |
   |                |  |                |
   +----------------+  +----------------+
```

Fetching the certificate, encrypting the configuration, publishing the encrypted configuration.

4.3. Initial Customer Boot

When the device is first booted by the customer (and on subsequent boots), if the device has no valid configuration, it will use existing auto-install type functionality - it performs DHCP Discovery until it gets a DHCP offer including DHCP option 66 or 150, contact the server listed in these DHCP options and download its config file.

After retrieving the config file, the device will examine the file and determine if it seems to be a valid config, and if so, proceeds as it normally would. Note that this is existing functionality (for example, Cisco devices fetch the config file named by the Bootfile-Name DHCP option (67)).

If the file appears be "garbage", the device will attempt to decrypt the configuration file using its private key. If it is able to decrypt and validate the file it will install the configuration, and start using it. The exact method that the device uses to determine

if a config file is "valid" is implementation specific, but a normal config file looks significantly different to an encrypted blob.

Note that the device only needs DHCP and to be able to download the config file; after the initial power-on in the factory it never need to access the Internet or vendor or certifcate publication server - it (and only it) has the private key and so has the ability to decrypt the config file.

```
             +--------+                +--------------+
             | Device |                |Config server |
             +--------+                | (e.g  TFTP)  |
                                       +--------------+
   +---------------------------+    +------------------+
   | +-----------+             |    |                  |
   | |           |             |    |                  |
   | |   DHCP    |             |    |                  |
   | |           |             |    |                  |
   | +-----+-----+             |    |                  |
   |       |                   |    |                  |
   | +-----v------+            |    |  +-----------+   |
   | |            |            |    |  | Encrypted |   |
   | |Fetch config|<------------------>|  config   |   |
   | |            |            |    |  |   file    |   |
   | +-----+------+            |    |  +-----------+   |
   |       |                   |    |                  |
   |       X                   |    |                  |
   |      / \                  |    |                  |
   |     /   \ Y    +--------+ |    |                  |
   |    |Sane?|---->|Install,| |    |                  |
   |     \   /      |  Boot  | |    |                  |
   |      \ /       +--------+ |    |                  |
   |       V                   |    |                  |
   |       |N                  |    |                  |
   |       |                   |    |                  |
   | +-----v------+            |    |                  |
   | |Decrypt with|            |    |                  |
   | |private key |            |    |                  |
   | +-----+------+            |    |                  |
   |       |                   |    |                  |
   |       |        +--------+ |    |                  |
   |       |        |Install,| |    |                  |
   |       +------->|  Boot  | |    |                  |
   |                +--------+ |    |                  |
   |                           |    |                  |
   |                           |    |                  |
   +---------------------------+    +------------------+
```

Device boot, fetch and install config file

5. Additional Considerations

5.1. Key storage

Ideally, the keypair would be stored in a TPM on something which is identified as the "router" - for example, the chassis / backplane. This is so that a keypair is bound to what humans think of as the

"device", and not, for example (redundant) routing engines. Devices which implement IEEE 802.1AR could choose to use the IDevID for this purpose.

5.2. Key replacement

It is anticipated that some operator may want to replace the (vendor provided) keys after installing the device. There are two options when implementing this - a vendor could allow the operator's key to completely replace the initial device generated key (which means that, if the device is ever sold, the new owner couldn't use this technique to install the device), or the device could prefer the operators installed key. This is an implementation decision left to the vendor.

5.3. Device reinstall

Increasingly, operations is moving towards an automated model of device management, whereby portions (or the entire) configuration is programmatically generated. This means that operators may want to generate an entire configuration after the device has been initially installed and ask the device to load and use this new configuration. It is expected (but not defined in this document, as it is vendor specific) that vendors will allow the operator to copy a new, encrypted config (or part of a config) onto a device and then request that the device decrypt and install it (e.g: 'load replace <filename> encrypted)). The operator could also choose to reset the device to factory defaults, and allow the device to act as though it were the initial boot (see Section 4.3).

6. IANA Considerations

This document makes no requests of the IANA.

7. Security Considerations

This mechanism is intended to replace either expensive (traveling employees) or insecure mechanisms of installing newly deployed devices such as: unencrypted config files which can be downloaded by connecting to unprotected ports in datacenters, mailing initial config files on flash drives, or emailing config files and asking a third-party to copy and paste it over a serial terminal. It does not protect against devices with malicious firmware, nor theft and reuse of devices.

An attacker (e.g a malicious datacenter employee) who has physical access to the device before it is connected to the network the attacker may be able to extract the device private key (especially if

it isn't stored in a TPM), pretend to be the device when connecting to the network, and download and extract the (encrypted) config file.

This mechanism does not protect against a malicious vendor - while the keypair should be generated on the device, and the private key should be securely stored, the mechanism cannot detect or protect against a vendor who claims to do this, but instead generates the keypair off device and keeps a copy of the private key. It is largely understood in the operator community that a malicious vendor or attacker with physical access to the device is largely a "Game Over" situation.

Even when using a secure bootstrapping mechanism, security conscious operators may wish to bootstrapping devices with a minimal / less sensitive config, and then replace this with a more complete one after install.

8. Acknowledgements

The authors wish to thank everyone who contributed, including Benoit Claise, Sam Ribeiro, Michael Richardson, Sean Turner and Kent Watsen. Joe Clarke provided significant comments and review.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

9.2. Informative References

[I-D.ietf-sidr-iana-objects]
Manderson, T., Vegoda, L., and S. Kent, "RPKI Objects issued by IANA", draft-ietf-sidr-iana-objects-03 (work in progress), May 2011.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC8572] Watsen, K., Farrer, I., and M. Abrahamsson, "Secure Zero Touch Provisioning (SZTP)", RFC 8572, DOI 10.17487/RFC8572, April 2019, <https://www.rfc-editor.org/info/rfc8572>.

Appendix A. Changes / Author Notes.

[RFC Editor: Please remove this section before publication]

From -00 to -01

- o Nothing changed in the template!

From -01 to -03:

- o See github commit log (AKA, we forgot to update this!)

- o Added Colin Doyle.

From -03 to -04:

Addressed a number of comments received before / at IETF104 (Prague). These include:

- o Pointer to https://datatracker.ietf.org/doc/draft-ietf-netconf-zerotouch -- included reference to (now) RFC8572 (KW)

- o Suggested that 802.1AR IDevID (or similar) could be used. Stress that this is designed for simplicity (MR)

- o Added text to explain that any unique device identifier can be used, not just serial number - serial number is simple and easy, but anything which is unique (and can be communicated to the customer) will work (BF).

- o Lots of clarifications from Joe Clarke.

- o Make it clear it should first try use the config, and if it doesn't work, then try decrypt and use it.

- o The CA part was confusing people - the certificate is simply a wrapper for the key, and the Subject just an index, and so removed that.

- o Added a bunch of ASCII diagrams

Appendix B. Demo / proof of concept

This section contains a rough demo / proof of concept of the system. It is only intended for illustration; presumably things like algorithms, key lengths, format / containers will provide much fodder for discussion.

It uses OpenSSL from the command line, in production something more automated would be used. In this example, the unique identifier is the serial number of the router, SN19842256.

B.1. Step 1: Generating the certificate.

This step is performed by the router. It generates a key, then a csr, and then a self signed certificate.

B.1.1. Step 1.1: Generate the private key.

```
$ openssl genrsa -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.................................................
.................................................
..........................+++
...................+++
e is 65537 (0x10001)
```

B.1.2. Step 1.2: Generate the certificate signing request.

```
$ openssl req -new -key key.pem -out SN19842256.csr
Country Name (2 letter code) [AU]:.
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:.
Organization Name (eg, company) [Internet Widgits Pty Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []:SN19842256
Email Address []:.

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:.
```

B.1.3. Step 1.3: Generate the (self signed) certificate itself.

```
$ openssl req -x509 -days 36500 -key key.pem -in SN19842256.csr -out
SN19842256.crt
```

The router then sends the key to the vendor's keyserver for publication (not shown).

B.2. Step 2: Generating the encrypted config.

The operator now wants to deploy the new router.

They generate the initial config (using whatever magic tool generates router configs!), fetch the router's certificate and encrypt the config file to that key. This is done by the operator.

B.2.1. Step 2.1: Fetch the certificate.

```
$ wget http://keyserv.example.net/certificates/SN19842256.crt
```

B.2.2. Step 2.2: Encrypt the config file.

I'm using S/MIME because it is simple to demonstrate. This is almost definitely not the best way to do this.

```
$ openssl smime -encrypt -aes-256-cbc -in SN19842256.cfg\
  -out SN19842256.enc -outform PEM SN19842256.crt
$ more SN19842256.enc
-----BEGIN PKCS7-----
MIICigYJKoZIhvcNAQcDoIICezCCAncCAQAxggE+MIIBOgIBADAiMBUxEzARBgNV
BAMMClNOMTk4NDIyNTYCCQDJVuBlaTOb1DANBgkqhkiG9w0BAQEFAASCAQBABvM3
...
LZoq08jqlWhZZWhTKs4XPGHUdmnZRYIP8KXyEtHt
-----END PKCS7-----
```

B.2.3. Step 2.3: Copy config to the config server.

```
$ scp SN19842256.enc config.example.com:/tftpboot
```

B.3. Step 3: Decrypting and using the config.

When the router connects to the operator's network it will detect that does not have a valid configuration file, and will start the "autoboot" process. This is a well documented process, but the high level overview is that it will use DHCP to obtain an IP address and config server. It will then use TFTP to download a configuration file, based upon its serial number (this document modifies the solution to fetch an encrypted config file (ending in .enc)). It will then then decrypt the config file, and install it.

B.3.1. Step 3.1: Fetch encrypted config file from config server.

```
$ tftp 192.0.2.1 -c get SN19842256.enc
```

B.3.2. Step 3.2: Decrypt and use the config.

```
$ openssl smime -decrypt -in SN19842256.enc -inform pkcs7\
  -out config.cfg -inkey key.pem
```

If an attacker does not have the correct key, they will not be able to decrypt the config:

```
$ openssl smime -decrypt -in SN19842256.enc -inform pkcs7\
  -out config.cfg -inkey wrongkey.pem
Error decrypting PKCS#7 structure
140352450692760:error:06065064:digital envelope
 routines:EVP_DecryptFinal_ex:bad decrypt:evp_enc.c:592:
$ echo $?
4
```

Authors' Addresses

Warren Kumari
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043
US

Email: warren@kumari.net

Colin Doyle
Juniper Networks
1133 Innovation Way
Sunnyvale, CA 94089
US

Email: cdoyle@juniper.net