wireless control for everyone

PREVENTING REPLAY ATTACKS AND OTHER SECURITY CONSIDERATIONS IN ONE-NET

Version 2.3.0

August 14, 2012

Table of Contents

1 Overview

All ONE-NET networks use a shared XTEA Encryption key. The network is considered insecure if this key is known outside of the network. This document discusses encryption key management, lost keys, when keys need to be changed, how keys are exchanged, and the definition and prevention of “replay attacks”.

2 Intended Audience

This document is geared towards both programmers and non-programmers. Certain sections will only be of interest to programmers. Network designers should also read this document to know what the issues are and what decisions need to be made. In addition, consumers and other end users should also read this document so that they are aware of the security concerns.

3 What Encryption Is Used In ONE-NET

ONE-NET uses the XTEA (Extended Tiny Encryption Algorithm) encryption technique to encrypt its packets. XTEA encryption uses a 128-bit key and encrypts data in 64-bit blocks. Hence all ONE-NET packets have encrypted payloads that are multiples of 64 bits in size. All ONE-NET packets are encrypted using this technique. All packets except for Stream Data packets use 32 rounds of encryption. Stream Data packets use 8 rounds of encryption. Therefore Stream Data packets should be considered less secure than other packets and therefore extremely sensitive information should not be sent in Stream packets.

Whenever there is the slightest concern that the network encryption key has been compromised, any device in the ONE-NET network can and should request that the master change keys. Keys are changed one 32-bit fragment at a time. Many applications will choose to automatically change keys at regular intervals even if there is no reason to believe that the key has been compromised.

4 How Are Keys Changed In ONE-NET

First, the master selects a new 32-bit key fragment to use. It will send that fragment out to each device using the old key. Each device, upon receiving the new key fragment, should delete the first 32 bits of the current key, slide the remaining 96 bits to the left, and append the new key fragment to the end, then

report back to the master using the new key. The master will keep track of which devices have reported back using the new correct key. When all devices have reported back using the correct key, the key change is complete. When choosing a new 32-bit key fragment, the new key fragment must not match any of the current 32-bit key fragments.

5 Complications When Changing Keys

Several things can go wrong when changing key fragments.

1. If two devices attempt other than the master attempt to communicate during the key change, and one device has the new key and one does not, it is likely that the two devices will be unable to communicate during this time.
2. Only one key fragment update can occur at once. A second key update can only occur after all devices get the first key update. This can be quite problematic when there are sleeping devices in the network which only check in once an hour or once a day. If more than one key update is requested when a device is sleeping, then the master must make a choice. Should the second key update be allowed to proceed or must it wait? If it waits, potentially there will be a time period where devices are forced to communicate with a possibly compromised key. If it proceeds with the second key change, the sleeping device may never be able to communicate again since it has lost the key.
3. Once a key has been compromised, simply sniffing all of the traffic in the air can allow an attacker to detect key change and update the key.
4. Several solutions to the problems above have been discussed, both in the core ONE-NET code and in the application level code, but none has been officially adopted. If one suspects that a dedicated attacker is sniffing the air and has the current key, currently the best way to handle this is to somehow pick a new 128-bit key and load it into each device in the network in a manner other than changing keys via the standard ONE-NET message types from the master.
5. Currently this scenario is considered quite rare. It is quite important for many network types, but not most.

6 Replay Attacks

A replay attack occurs when an attacker does not the know the encryption key, sniffs the air, detects a legitimate command, then "replays" the legitimate command at a later time. The most common form of this is for an attacker to wait with a sniffing device in a car outside of a garage or an office and wait for someone to open the door via remote control. The attacker figures out which message opened the door and tries the exact same message an hour later. If countermeasures are not taken, that message will open the door again.

To prevent this scenario, in a secure system, no legitimate single command that can open a lock should be used twice. Either the encryption key and / or the message contents must change.

Please also see section 4.2.9.1.2 for more information on Message IDs and Replay Attacks.

ONE-NET solves this basic problem by creating a Message ID field. Suppose that device 002 is a garage door opener and device 004 is a relay controlling a garage door. Device 002 sends an "on" command to device 004. Assuming the message is valid, the garage door will open.

Let's look at a potential message that could do this. Let's assume that a state of "On" means "Open the door" and a state of "Off" means "Close the door". Let's assume that the garage door unit and the remote control unit are both 0.

The message would be as follows…

1. This is a single, 1 block application message.
2. This application message will use the normal parsing mechanism (ON_APP_MSG).
3. The message class will be "command".
4. The source unit will be 0.
5. The destination unit will be 0.
6. The message type will be "Switch".
7. The message data will be "On".

Putting this all together, we'll have 64 bits of payload, coded as follows…

Pld. CRC 8 BITS	Message ID 12 BITS	Message Type 4 BITS	Transaction Data Payload 40 BITS

Filling in the Message Type, we have “0” for “ON_APP_MSG”.

Pld. CRC 8 BITS	Message ID 12 BITS	Message Type 0000	Transaction Data Payload 40 BITS

Recall that the parsing mechanism for ON_APP_MSG is as follows…

Message Class 4 BITS	Message Type 8 BITS	Source Unit 4 BITS	Dest. Unit 4 BITS	Message Data 20 BITS

We can fill those in now. ONA_COMMAND is 5, or 0101 in bits.

Message Class 0101	Message Type 8 BITS	Source Unit 4 BITS	Dest. Unit 4 BITS	Message Data 20 BITS

The message type is “switch”. ONA_SWITCH is 0, or 00000000 in bits.

Message Class 0101	Message Type 00000000	Source Unit 4 BITS	Dest. Unit 4 BITS	Message Data 20 BITS

The source and destination units are both 0, so we can fill those in too.

Message Class 0101	Message Type 00000000	Source Unit 0000	Dest. Unit 0000	Message Data 20 BITS

Finally, the message data is “1”, or “ONA_ON”. We’ll fill that in.

Message Class 0101	Message Type 00000000	Source Unit 0000	Dest. Unit 0000	Message Data 00000000000000000001

All in all we have 64 bits of payload to fill in. The 40 bytes of transaction data payload will be the same each time. The 4 message class bytes will be the same each time. The 8 bit CRC field is completely determined by the other 56 bytes. The only thing that varies are the 12-bit Message ID, so we have 4096 valid "open the garage door" messages. We have to assume that any one of them can be intercepted and replayed. We need to make sure that does not work, so none of them can be repeated. Hence we can at most open the garage door 4096 times before we have to change the key fragment.

Here is the way we manage this, ideally. Devices 002 (the garage door opener) and 004 (the garage door), when they first communicate, must exchange features. They must also establish a Message ID. When device 002 first attempts to open 004's door, it will pick a random low Message ID to use. Device 002 will reject that Message ID and pick a Message ID that is slightly higher, NACK the message, then send the acceptable Message ID back with the NACK. Device 002 should try again, but this time use the Message ID that was passed to it.

All subsequent commands from device 002 to device 004 will increment the Message ID. Thus device 004 always expects a higher Message ID than it got last time. It will reject any Message ID that is NOT higher than the one that is on its table. Thus any previously intercepted message will not work. When device 002 or 004 sense that they are about to run out of Message IDs, they must request a key change from the master device so that no message will ever be repeated using the same key and Message ID.

A replay attack is thus thwarted as follows. We'll assume, as always, that the good guys are named Bob and Alice and the person trying to break in is named Eve. Bob and Alice are a married couple who have set up a small ONE-NET network consisting of device 002 (Bob's remote control), device 003 (Alice's remote control), and device 004 (the garage door opener). Eve is the nosy, electronically sophisticated next-door-neighbor / eavesdropper (hence the name Eve) who has set up a sophisticated sniffing device.

1. 9 a.m. – Bob sends an "open the door" command from 002 (remote control) to 004 (garage door opener) using Message ID 0x090. The door opens. Device 002 and 004 both store the value 0x090. Eve intercepts the message.

2. 10 a.m. – Bob sends an "open the door" command from 002 (remote control) to 004 (garage door opener) using Message ID 0x091. The door opens. Device 002 and 004 both store the value 0x091. Eve intercepts the message.
3. 11 a.m. – Bob sends an "open the door" command from 002 (remote control) to 004 (garage door opener) using Message ID 0x092. The door opens. Device 002 and 004 both store the value 0x092. Eve intercepts the message.
4. Noon – Eve decides to try repeating her messages to see if any of them work. Let's say she tries the last one. Eve sends an "open the door" command from her own device that is not part of the network. Eve pretends to be Bob's garage door opener (002) and replays the last intercepted message that worked. That message contains Message ID 0x092. Device 004 looks up device 002 on its table and notes that the Message ID is 0x92. Valid Message IDs must be LARGER than the record in the table. Hence 004 refuses to open the garage door. It sends back a NACK message and attaches the value 0x093 to the message. That is the Message ID it WILL accept.
5. If the device sending the message was truly Bob's garage door, it would know the encryption key and be able to decipher the NACK message, repackage the message using the correct Message ID, and try again. However, Eve's device does NOT know the key, so it cannot decipher the NACK and try again using the correct Message ID. Eve's attempt has been foiled!

6.1 Potential Problem 1 With The Solution Above

The solution above requires that device 004 never "loses track" of device 002 and the Message ID that was last used. Recall, however, that clients can only keep track of a certain number of other clients, and that value is set by the value of ONE_NET_RX_FROM_DEVICE_COUNT (see the **ONE-NET Configuration Options, Features, And Port Constants Guide** document). This table maintains a "Least Recently Used" Scheme. If too many devices communicate with device 004, Bob's device could fall off of the table and device 004 could therefore lose track of the Message ID of Bob's garage door opener and accept one that was previously accepted. Eve could get lucky, or if Eve could send an intercepted route message or a features request message or some other intercepted message from a variety of devices in an attempt to FORCE device 002 to "fall off" the table. After it does, Eve can replay an earlier message and perhaps get lucky.

6.2 "Slideoff" Settings

A solution to the problem above is to set the "slideoff" setting for the devices that legitimately open doors. ONE-NET provides an enumerated type called device_slideoff_t, defined in the file one_net3_message.h. Using this option, certain devices, once on the table, can be made immune to "sliding off" the devlice table using the "Least Recently Used" algorithm. An easy solution to the problem defined above in section 6.1 is for device 004 (the garage door opener) to use this feature to prevent any device that has successfully opened it previously as a security measure.

6.3 Potential Problem 2 With The Solution Above

ONE-NET provides for methods of preventing "slideoff" and preventing duplicate Message IDs and these are enough for most needs. However, Eve has a few more tricks up her sleeve that must be guarded against. Consider the following scenario.

1. 9 a.m. – Bob sends an "open the door" command from 002 (remote control) to 004 (garage door opener) using Message ID 0x090. The door opens. Device 003 and 004 both store the value 0x090. Eve intercepts the message.
2. 10 a.m. – Alice sends an "open the door" command from 003 (remote control) to 004 (garage door opener) using Message ID 0x091. The door opens. Device 003 and 004 both store the value 0x091 in their tables. Eve intercepts the message.
3. 11 a.m. – Eve has two intercepted messages, one from Alice, one from Bob. She tries them both.
    a. She attempts to send a message pretending to be device 002 using Message ID 0x90. Device 004 checks its table and notes that the last message from 002 used Message ID 0x090. Thus THIS TIME the Message ID should be higher. 004 therefore rejects the message.
    b. She attempts to send a message pretending to be device 003 using Message ID 0x91. Device 004 checks its table and notes that the last message from 003 used Message ID 0x091. Thus THIS TIME the Message ID should be higher. 004 therefore rejects the message.

c. Thus Eve has intercepted Bob's message and replayed it. Eve has intercepted Alice's message and replayed it. Neither attempt worked due to invalid Message IDs.

d. Now Eve switches things up. She repackages Alice's intercepted message and sends it, pretending this time not to be Alice, but instead pretends to be Bob by repeating Alice's message, but changing the source and repeater DIDs from 003 to 002 so that it appears that the message is from Bob.

e. This attempt will work. The Message ID is 0x91. The garage door opener, believing that the message was sent from Bob (device 002), will look in its table and find 002. The Message ID in the table is 0x90 for 002. The Message ID in this message is one more than the one in the table, exactly what the garage door opener expects for a LEGITIMATE message from BOB. The garage door opens. Eve has succeeded!

6.4 Solutions To The Problem Above

Currently ONE-NET does not implement either of the solutions above directly. The application code needs to do it. Future additions may incorporate these techniques.

1. Solution 1 – Device 004 should always demand that the Message ID that is received is higher than the Message ID of ANY device on its table (including the Master device, just to be safe). Or 004 simply keeps track of ONE Message ID for the entire network and rejects any Message ID less than or equal to the one on file. This results in less memory consumption, more retries are required to sync Message IDs, but the result is that, one, "slideoff" replay attacks are no longer a factor, and two, replay attacks like the scenario above are no longer possible.
2. Solution 2 – Certain Message Types other than ON_APP_MSG can be used for these messages. The ON_APP_MSG parsing type reserves 20 bits for the message data. A large percentage of replay attacks are "on/off" messages that uses only one of the twenty bits. A new parsing mechanism can be put in place like this…

Message Class 4 BITS	Message Type 8 BITS	Source Unit 4 BITS	Dest. Unit 4 BITS	Raw Source DID 12 BITS	Message Data 8 BITS

If the Source DID in the packet address does not match the Source DID in the payload, the message is rejected. In the scenario above, since Alice's message was replayed, her remote control's Source DID (003) would be stored in the 12 bit Raw Source ID field, which would not match Bob's remote control DID (002) that Eve was pretending to use, so the message would not be accepted. 8 bits for message data is plenty large enough to hold the Message Data for an on/off command.