OpenPAYGO

EnAccess

Open Source PAYGO Token

Security Audit

PaygOps
Last-Mile Management System

The EnAccess Foundation

Approval Process

This audit was first realised by Solaris Offgrid and was then reviewed, amended and improved by Gábor Ács-Kurucz, a Security Architecture Specialist who is not affiliated with Solaris Offgrid.

	Name ***Affiliation***	**Date**	**Signature**
Written by	Benjamin David *Solaris Offgrid*	30 Aug 2019	[illegible]
Reviewed by	Gábor Ács-Kurucz *Independent Security Architecture Specialist*	26 Sep 2019	[illegible]

Index

Introduction

The token system has been designed for convenience and ease of use by base of pyramid consumers in developing countries. The assets to be locked using this system are likely to range in value from ~10$ to 1000$.

Most of the final users of the assets do not have internet connections at home nor access to sophisticated tools, but most can still get access to laptops or smartphones.

Considering that, security trade offs have been made to keep the tokens short while keeping any attack significantly more expensive than the assets protected by the system.

Here we review potential ways to attack the OpenPAYGO token system and how to protect against them, as well as some hardware attack that can affect any Pay As You Go system and for which implementing mitigation strategy can be important (depending on the asset value).

Software Attack Vectors Review

Dumb Token Bruteforce (Risk: Low)

With full knowledge of how the system works, and the will to generate one particular token (e.g. an unlock forever token), the number of potential combinations can be reduced to 999,999.

Out of those combinations, 30 are going to be valid (due to the maximum count jump), meaning that the chance of any randomly entered token being correct is 30 / 999,999 or 1/33,333.

With the recommended entry rate limiting (1 minute wait for the first invalid token entry, doubling with each subsequent one with a cap at 512 minutes), the first 10 tokens can be entered in 511 minutes, and each subsequent token can be entered every 512 minutes. This means that to have a 100% chance of getting a correct token, it will take over 32 years, while it would take ~16 years to have a 50% chance and still more than 3 years for a meagre 10% chance at success. This is all assuming the entry system (keypad, IR, etc.) is modified and replaced by an automatic entry device, manual entry would realistically be a lot longer.

Considering the lifespan and cost of the product typically protected by those mechanisms, this attack is not a practical solution.

Mitigation: The main mitigation method is ensuring that the waiting time in between incorrect token tries grows quickly to a relatively high time, and cannot be bypassed in any way (e.g. disconnecting the battery, forcing a product reset, etc.). The second mitigation is keeping the maximum count jump low, with 30 being the recommended value, but a lower value as low as 10 can be used to increase security (at the expense of more risk of having operational issues as described in the general documentation).

Statistically Improved Token Bruteforce (Risk: Low)

Statistical analysis on 10,000,000 tokens generated for different 10,000 different "devices" (with unique keys and starting code) shows that the variance is extremely high and that the only pattern seems to be related to the conversion from 30 bits to 29.5 bits that makes some higher numbers more common.

In particular, token values between 926,258,825 and 999,999,999 can be up to twice as likely as other values. If the searched token happens to be in that range which happens ~7% of the time), the likelihood of bruteforce success can be increased slightly. It could make the attack quicker if starting with those tokens and the searched token happening to be in that range. However in practice this is far from enough to make bruteforcing practical.

Mitigation: Same mitigation as the token brute force as this is just a way to increase its efficiency.

Synchronisation Token Bruteforce (Risk: Low)

While synchronisation tokens are convenient, it affects the level of security of the device by having a larger range of valid tokens (especially if the value of count tried are larger than the one suggested) and more likelihood of having a significant impact on activation. This means that it is a bit easier to "guess" or bruteforce a counter synchronisation token than a regular token, and some guessed counter synchronisation tokens (any of the 31 below the current device count) can allow the client to re-use older tokens (any with a count that is above that).

While for regular tokens, 30 are going to be valid, for reset tokens this can jump to 130 (if following guidelines) meaning that the chance of any randomly entered token being correct is 130 / 999,999 or 1/7693.

With the recommended entry rate limiting, to have a 100% chance of getting a correct token, it will take ~7.5 years, while it would take ~3.75 years to have a 50% chance and still more than 9 months for a meagre 10% chance at success. This is all assuming the entry system (keypad, IR, etc.) is modified and replaced by an automatic entry device, manual entry would realistically be a lot longer.

Mitigation: Reducing the range of allowed counter sync greatly reduce this risk and is recommended for higher value devices. Having an increased waiting period (e.g. 1024 minutes) for high value devices implementing counter sync is also recommended.

Starting Code Bruteforce (with known key and tokens) (Risk: Medium)

If the same key is used for many devices (e.g. all from one manufacturer) and the key happens to leak, bruteforcing the starting code of a particular device can then be practical if a few tokens are known with their count and value.

Such token with known count and value can reasonably be obtained by an end user just by deduction, he knows the number of days he pays for and if the count starts at 1 he can deduce the count based on the order of his payments.

By going through all possible starting codes until the token generated with them with the specific value and count matches the known ones, one needs only ~2 million tokens to be generated to be able to get the starting code. This can be achieved quickly on a regular computer for a low cost.

This attack is overall quite realistic, but the required condition of having the key leaked is quite unlikely. However, it is not impossible that the key is found, especially if a manufacturer uses a single key for a large number of products, making a hardware attack (see below) economically viable to get the key on one of the products and then use this method on other products with the first few token given to the client at registration and a dedicated mobile app, to then generate a full unlock code for the product.

Mitigation: The main mitigation is ensuring that keys are always kept safe (at the manufacturer's facility, during exchange with the server, on the server and on the devices themselves). However, even with those precautions, recovering a key from a device is usually possible with hardware methods (see below). This is however quite expensive and not economically viable for one device, so the best mitigation strategy is to use different key for each device, or when impractical at least different key for each small batch of devices.

Key Bruteforce (with known starting code and tokens) (Risk: Very Low)

If the starting code of a device can be known (e.g. printed on a sticker or included on the barcode) then a bruteforce of the key is in theory possible if a few tokens are known with their count and value (see Starting Code bruteforce on how those can be obtained).

In the best case, a specialised ASIC and a token with count 1, one might be able to get up to ~100 million tokens per second for 100 USD with a lifetime of 10 years. This would give a cost of 3x10^-7 USD per second or 3x10^-15 USD per hash tried. To find the 128 bits key, that requires trying ~3x10^38 hashes, it would hence cost up to ~9x10^23 USD or 900,000,000,000,000,000,000,000 USD to be sure to find an unlock token given a few activation tokens with known value and count. The cost to get a 10% chance at success is still almost 10^23 USD. This does not include electricity costs which should be in the same order of magnitude. This computation would likely generate a few colliding keys, which would require testing a few codes onto the device.

Overall this attack does not seem economically viable at all considering that the cost of the attack is tens of orders of magnitude higher than the price of the protected assets.

Mitigation: The attack can be mitigated by giving a random starting count between 0 and 10 making it harder to guess the count values of tokens given to end user and making the token calculation more computationally expensive (with several cycles of hash per token). It is also critical to ensure that keys are properly randomly generated and cannot be guessed more easily.

Hardware Attack Vectors Review

These hardware attacks are not directly linked to OPAYGO but they can affect any device using a PAYGO token system and understanding their risk and mitigation strategy is key to implementing safe PAYGO.

External MCU Communication Hack (Risk: High)

To make the PAYGO integration easier or due to memory constraints on the device's main MCU, some systems use an external MCU to perform the PAYGO function. The way the PAYGO status is signaled to the main MCU doing the control of the device is then a potential attack vector, especially because the signaling is often done just with a HIGH or LOW signal on a line.

This makes an attack relatively easy by simply modifying that line to change the PAYGO status of the device, often by just soldering a wire between two pads (VCC and PAYG_ENABLE or GND and PAYG_ENABLE). However, this type of hardware attack needs to be performed on each device to hack and requires some level of knowledge, skills, and tools that might not be available to most of our clients. Nonetheless, it can be a very realistic attack for products that have a higher value.

Mitigation: The strongest mitigation is implementing the PAYGO function directly in the main MCU controlling the function of the device. When that is not possible, implementing a more secure way of signaling the PAYGO status to the main MCU is highly recommended. Ideally, this would a protocol with some security, but when memory or computational constraints are too tight on the main MCU, a simple PWM output with a given ratio that can easily be read by the main MCU already makes the attack significantly harder by requiring some additional hardware to be added to generate the signal. Ideally the PWM ratio can even vary between devices to make the larger scale production of "modchips" less economical.

PAYGO Switch Hack (Risk: High)

In many cases, the function of the device and its ON/OFF status is controlled by a limited number of switched (relays or FET) with a simple HIGH or LOW control. In that case, just switching those switches by rewiring them can change the status of the device.

However, depending on the device complexity, this might only provide partial function of the device. Since this requires some skilled work and needs to be performed on each device, the risk is highest for high value devices.

Mitigation: Have sensors measuring the output that can detect this kind of attack, log them, and disable key functions. Make sure some of the critical features require the MCU to work.

Slow Clock Hack (Risk: Medium)

The PAYGO status is reliant on the clock that the MCU is using, and tricking that clock into staying still can trick the system into staying active forever, instead of expiring after the days given by the token. Depending on the hardware, this can be achieved in several ways.

For system using an external oscillator or crystal (usually 32.768kHz), for some MCU and some RTC integrated circuits, just removing the crystal causes the clock to stay still. One could also generate a much slower signal on those lines.

For systems using an RTC integrated circuit outside of the MCU, reprogramming that RTC or replacing it with a “fake” RTC outputting a constant time could be used as an attack.

Mitigation: The MCU should frequently compare the external clock source to an internal clock source reliant on no external components to make sure the clock is moving at the expected speed.

MCU Key readout (Risk: Medium)

Nearly all MCUs are vulnerable to reading out the content of their memory. For MCUs without readout protection, this can be relatively simple and cheap, especially if some type of debug interface (e.g. JTAG) is exposed or has a connector, one can devise a small handled device that would generate key when connected to the device.

However, depending on the MCU this can be more or less practical, some having fairly solid readout protection and requiring expensive dedicated equipment to extract the data which can be extremely expensive.

The main risk for this attack if for it to be used in combination with a Starting Code bruteforce if the same key is used for a large quantity of product, in which case even a relatively expensive key extraction can be economical.

Mitigation: Making sure different keys are used for each device to make the attack less interesting. Also ensuring to use MCUs with good readout protection.

Accelerated Token Bruteforce (Risk: Medium)

Depending on the hardware, it can be possible to accelerate a token brute force by reducing the waiting periods. This can work in a similar way to the Slow Clock attack but in reverse, by making the clock go faster than designed, one can significantly reduce the waiting periods in between token entry and make a Token brute force attack more likely.

Mitigation: The MCU should frequently compare the external clock source to an internal clock source reliant on no external components to make sure the clock is moving at the expected speed. Alternatively, only the internal clock can be used for waiting periods, since modifying it to make it quicker would require more extreme measures (extreme cooling, overclocking, etc.).

Alternative Firmware Flashing (Risk: Low)

To circumvent the PAYGO locking, an attacker can try flashing an alternative firmware onto the MCU, that will keep the load switch on. Depending on the complexity of the device this can sometimes be easy, while it can be a project requiring several engineering months on other hardware if many features rely on the MCU.

Mitigation: Ideally, use one time programmable MCUs. When not possible, avoid exposing programming ports to make the flashing task harder. You can also make sure some of the critical functions are performed by the MCU and that they are not easy to replicate. For example, putting the battery charge algorithm or some communication on the MCU will make recreating an alternative firmware costly. Ensuring that reading out the original MCU firmware is difficult is also important.

Conclusion

Overall the token system is perfectly safe for its purpose and practical attacks are very unlikely. As long as basic care is taken to protect the keys, software attacks should not be a worry. The main risk factors are related to the actual device on which the token is used, following the recommendations and implementing some mitigation strategy can go a long way towards greatly reducing the risks of attack. Since hardware attack usually require a skilled technician modifying the device, they are extremely unlikely for low value devices but can become more realistic for high value devices for which we recommend an increased investment in mitigation.

HELLOSIGN

Audit Trail

TITLE	OpenPAYGO Token Security Audit
FILE NAME	Security Audit (5).pdf
DOCUMENT ID	89b563dbbed9a913cbca17b52f2c36dabe839ae0
STATUS	● Completed

Document History

SENT	**25** / **09** / **2019** 16:58:28 UTC+1	Sent for signature to Benjamin David (benjamin@solarisoffgrid.com) and Gábor Ács-Kurucz (gabika92@gmail.com) from siten@solarisoffgrid.com IP: 139.47.112.153
VIEWED	**25** / **09** / **2019** 16:58:50 UTC+1	Viewed by Benjamin David (benjamin@solarisoffgrid.com) IP: 139.47.112.153
SIGNED	**25** / **09** / **2019** 16:59:01 UTC+1	Signed by Benjamin David (benjamin@solarisoffgrid.com) IP: 139.47.112.153
VIEWED	**25** / **09** / **2019** 23:46:56 UTC+1	Viewed by Gábor Ács-Kurucz (gabika92@gmail.com) IP: 213.55.220.29
SIGNED	**26** / **09** / **2019** 14:14:24 UTC+1	Signed by Gábor Ács-Kurucz (gabika92@gmail.com) IP: 213.55.221.149
COMPLETED	**26** / **09** / **2019** 14:14:24 UTC+1	The document has been completed.