ARMIS

TLStorm

Critical vulnerabilities in a TLS library lead to complete pwnage of a popular Cloud-connected UPS

Gal Levy
Yuval Sarel
Ben Seri
Barak Hadad

Table of Contents

Introduction

Armis labs discovered three critical vulnerabilities affecting the *Smart-UPS* product line by APC (a subsidiary of Schneider Electric). Dubbed TLStorm, the set of vulnerabilities include two critical vulnerabilities in the TLS implementation used by Cloud-connected Smart-UPS devices, as well as a third vulnerability, a design flaw, in which firmware upgrades of all Smart-UPS devices are not properly signed and validated.

The two vulnerabilities that affect the TLS implementation are remote-code-execution (RCE) vulnerabilities and were found to be a result of improper use of a popular embedded TLS library (Mocana nanoSSL) which powers the devices' connection to APC's Cloud. An attacker that is able to establish a man-in-the-middle position can exploit these two vulnerabilities to target organizations directly over the Internet.

The third vulnerability is a fundamental design flaw in which the firmware files of Smart-UPS devices are not digitally signed, which can allow attackers to forge malicious firmware files, and install them on target devices via various paths. This can allow attackers to establish long-lasting persistence on such UPS devices that can be used as a stronghold within the network from which additional attacks can be carried.

APC is one of the market leaders in backup power suppliers, with over 20 million Smart-UPS units sold, meaning these vulnerabilities are likely to affect millions of organizations. By exploiting TLStorm, an attacker can breach internal networks from the Internet and perhaps more importantly, physically attack the organization's power lifeline.

This document will detail the inner workings of modern backup power systems, the attack surfaces they expose, and the discovered vulnerabilities. This document will also explore the risk encompassed in cyber-physical attacks that target popular and benign-looking devices such as network-connected or even Cloud-connected backup power systems.

Who we are

Armis Labs is the Armis research group and is focused on mixing and splitting the atoms that comprise the IoT devices that surround us - be it a smart personal assistant, a benign-looking printer, a SCADA controller, or a life-supporting device such as a hospital bedside patient monitor.

Our previous research includes:

- PwnedPiper - Nine vulnerabilities in Swisslog Pneumatic Tubes System (PTS) - a critical infrastructure used in hospitals
- Modipwn - Authentication bypass leads to remote-code-execution in Schneider Electric Modicon PLCs
- NAT Slipstreaming 2.0 - A NAT bypass technique that abuses support for VoIP protocols by NATs
- EtherOops - Exploit utilizing packet-in-packet attacks on ethernet cables to bypass firewalls & NATs.
- CDPwn - Five critical vulnerabilities in various implementations of the Cisco Discovery Protocol.
- URGENT/11 - 11 Zero-Day vulnerabilities impacting VxWorks, the most widely used Real-Time Operating System (RTOS).

Background

An Uninterruptible Power Supply (UPS) is a device that ensures zero downtime for downstream devices. Users connect mission-critical devices or devices which have sensitive roles (such as servers or medical devices) to the UPS as a power supply. The UPS in turn is connected to the power grid and relays the power to its downstream devices. The UPS contains a large battery, charged from the main power supply, so during a power shortage, it can instantaneously switch over its power source, and power the downstream devices (as well as its own circuitry) from the on-board battery. Since batteries are DC (direct current), and most appliances use AC (alternating current), the UPS has to invert the power, and produce a steady sinusoidal alternating current to its downstream devices. Doing so in a reliant and efficient manner is what requires UPS devices to be 'Smart', and controlled by embedded systems with complex electronics.

APC Smart-UPS

According to APC, the Smart-UPS line is the "ideal UPS for servers, point-of-sale, routers, switches, hubs, and other network devices", and has "over 20 million units sold".

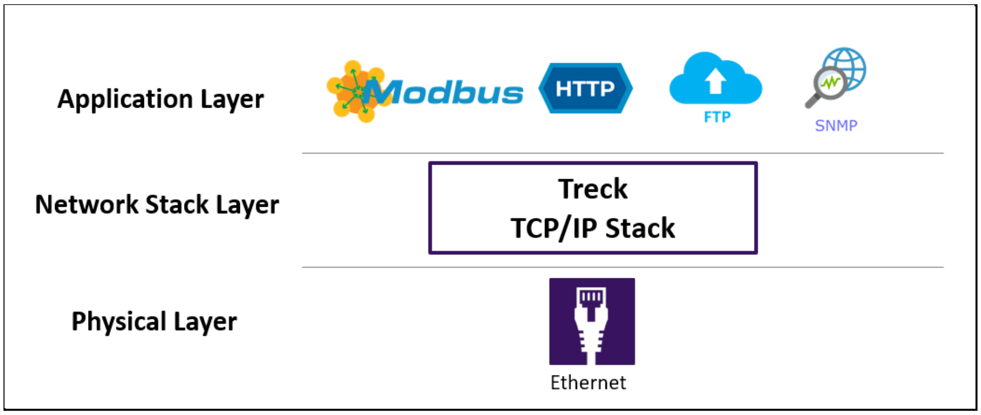
The device is indeed "smart" in various ways, such as implementing outlet groups and offering various methods of remote monitoring the performance and status of the UPS and its battery. These features can be accessed either directly through a panel on the device itself or via the network. The latter option has two variations:

1. Smart-UPS devices have a slot for a proprietary Network Management Card (NMC), which can be connected to the local network like any other network entity, and supports various network connectivity applications (web server, SNMP server, Modbus client over TCP, etc).
2. New-generation Smart-UPS models implement a feature called *SmartConnect*, which is a dedicated Ethernet port through which the device will connect to APC's Cloud service and allow remote management of the device.

Smart-UPS, front and back, NMC as external extension

Attack surface via Network Management Card (NMC)

The widest network attack surface of a Smart-UPS is through the NMC, as it implements multiple network applications:



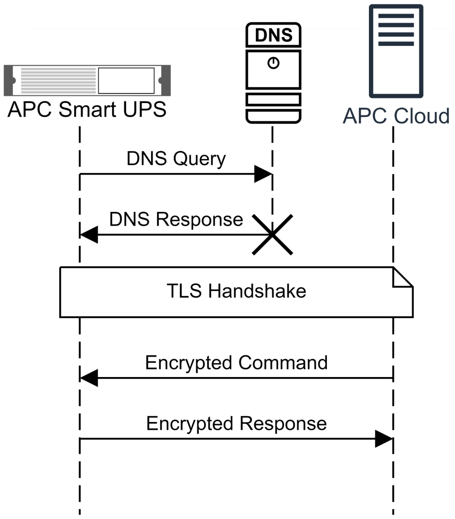
Network attack-surface of an NMC

The TCP/IP stack used by the NMC is Treck, which was discovered a few years ago to contain several vulnerabilities. Despite this, the entirety of the network attack surface (including Treck) that is exposed by the NMC only impacts the NMC itself -- the network applications do not run on the main processor of the UPS. Gaining code-execution on the NMC would still require a second phase to take over the main processor, and alter any logic it is responsible for.

However, the *SmartConnect* feature is implemented in the main processor, and poses a greater risk to the UPS as it operates over an **Internet connection** to the APC Cloud.

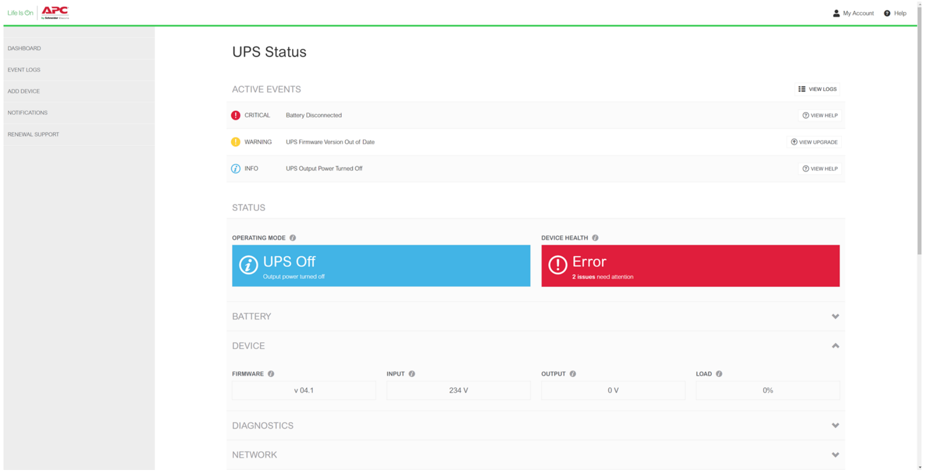
SmartConnect Cloud communication

The SmartConnect feature is implemented over a secure TLS connection to the APC Cloud, initiated by the UPS. The UPS resolves the domain of the APC Cloud service via DNS, and connects to it via TLS:



Simplified flow of a SmartConnect connection

Once the UPS connects to APC's Cloud, the user can log into their account and control their UPS. Supported features include monitoring various power consumption and battery statistics, and one notable feature is the remote firmware upgrade feature that can be used to install a new firmware over the cloud connection.



UPS Status

ACTIVE EVENTS

CRITICAL Battery Disconnected

WARNING UPS Firmware Version Out of Date

INFO UPS Output Power Turned Off

STATUS

OPERATING MODE: UPS Off
Output power turned off

DEVICE HEALTH: Error
2 issues need attention

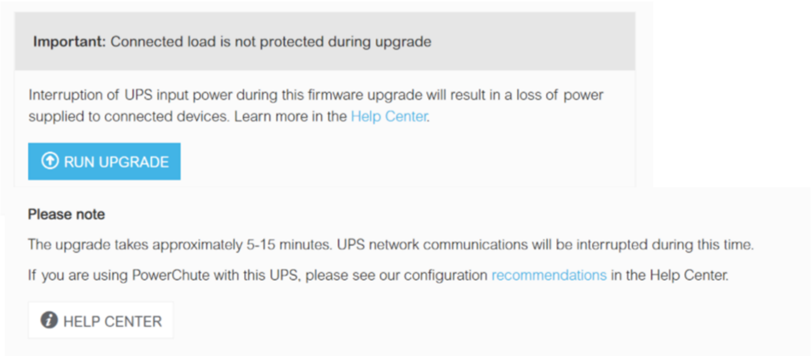
DEVICE

FIRMWARE	INPUT	OUTPUT	LOAD
v 04.1	234 V	0 V	0%

APC Cloud Interface

Remote firmware upgrade

Upon a successful connection, the APC Cloud service queries the UPS's model and firmware version. If a newer version for the specific model is available, the service automatically uploads the firmware to the UPS, allowing the user to actively finalize the installation process via the cloud console.



Important: Connected load is not protected during upgrade

Interruption of UPS input power during this firmware upgrade will result in a loss of power supplied to connected devices. Learn more in the Help Center.

Please note

The upgrade takes approximately 5-15 minutes. UPS network communications will be interrupted during this time.

If you are using PowerChute with this UPS, please see our configuration recommendations in the Help Center.

Firmware upgrade notification in the APC Cloud Interface

Once the user approves the update, the service instructs the UPS to install the new firmware and reboot.

As we'll detail in the next section, one of the discovered vulnerabilities can allow an attacker to impersonate the cloud service, while a second vulnerability can enable him to forge malicious firmware files. Combining the two with the remote firmware upgrade feature can allow an attacker to reach persistent code execution on target UPS devices from the Internet - **without neither authentication nor user interaction**.

Before diving into the vulnerabilities, a quick recap of the TLS protocol is needed.

TLS overview

Transport Layer Security (TLS) is a widely known and used cryptographic protocol. It is used in a variety of applications and protocols, and most notably in HTTPS - the secure extension of HTTP. It is also used to secure email protocols, voice over IP, network administration, and countless other applications that require secure and authenticated communication.

TLS connections have two phases:

- The handshake phase, in which the server and client agree on a cipher suite and exchange keys. Using the agreed upon keys they encrypt and decrypt the connection data.
- The application phase in which application data is encrypted, sent, and decrypted using the suite and keys exchanged during the TLS handshake.

In a standard HTTPS connection (the most common use of TLS) the TLS handshake will most often include a server sending an SSL certificate to the client, which the client will use to authenticate the server. In most TLS connections, the connection starts with an exchange of keys that is used to derive a temporary key with which the session will be encrypted.

mTLS

In the classic scenario of HTTPS, only the client is authenticating the server - a web server wants to serve any client that connects to it, but the clients want to validate the authenticity of the web service. Certain applications that utilize TLS require both parties to authenticate one another (a.k.a mutual TLS, or mTLS), and TLS offers a few options on how that can be achieved. The most straightforward approach is having the client also supply his own certificate during the TLS handshake, which the server will authenticate.

TLS-PSK

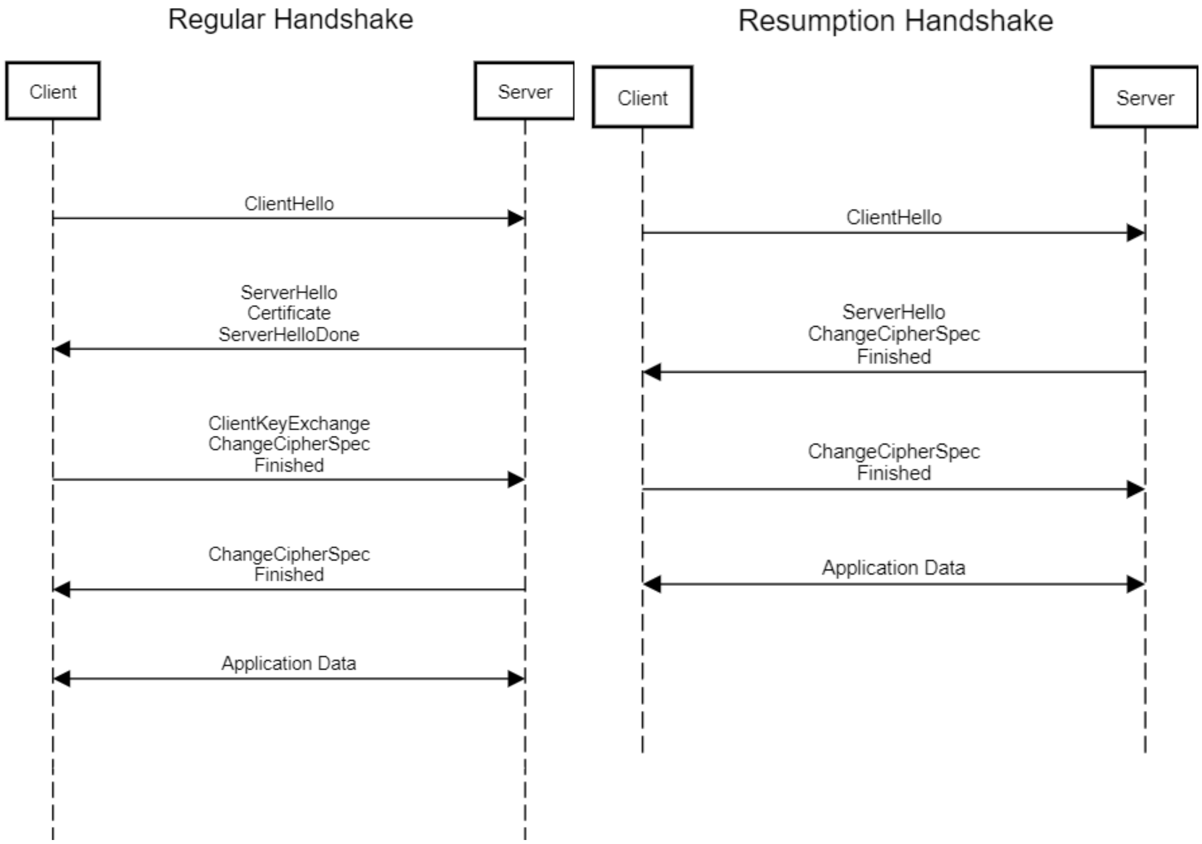
However, TLS supports other, much less common key exchange and authentication algorithms, such as TLS-PSK (TLS Pre-Shared-Key), where both parties have a pre-shared secret distributed to them before the session takes place. This secret is a symmetric key, used to both authenticate the entities (both server **and** client), and also to derive the temporary encryption keys.

Pre-shared keys can be an alternative form of mutual TLS (mTLS) and may offer a convenient method for certain applications, when a server expects connections only from specific known devices that have been bootstrapped in advance with the shared PSK.

A nominal use of TLS-PSK requires that the server holds a dictionary of keys, per client, thus having a unique key used for each client - and eliminating the possibility that a compromise of a single key may be used to undermine all the server's communications. Despite this, the use of TLS-PSK holds various additional security risks. For example, if an attacker manages to exfiltrate a pre-shared key from one of the server's clients, he can pose as the legitimate server in a man-in-the-middle attack.

TLS session resumption

An equally obscure feature of TLS is the TLS session resumption feature. This feature facilitates a quick resumption of an established TLS session that has unexpectedly dropped. A session ID is assigned to each connection once the TLS handshake is completed, which allows jumping back into the application phase using the same algorithm and secret from the previous session. The session ID may be included by the client in the *ClientHello* message or alternatively by the server in the *ServerHello* message - the initial messages in the TLS handshake. The session ID is stored by both parties along with the chosen encryption algorithm and the generated master secret, and can be used when TLS session resumption is initiated for session restoration.



Regular TLS handshake vs a Resumption TLS handshake

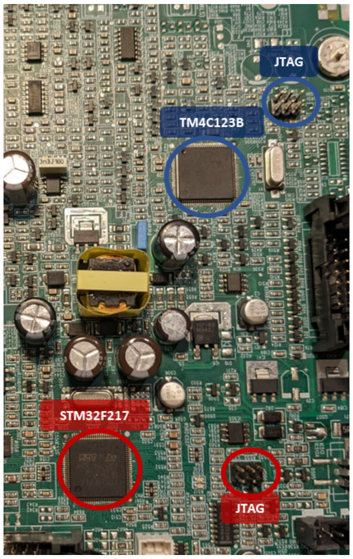
Vulnerability Hunting - Firmware Extraction

First step for hunting vulnerabilities is gaining access to the target's firmware, in order to analyze and reverse-engineer it. In this instance, the Smart-UPS proved to be a worthy opponent, and this first step turned out to be a challenging task.

Firmware Mitigations

Basic analysis of the firmware update files (available through APC's upgrade wizard) revealed they are encrypted. Analysis of the PCB of the SMT750I device (one of the small SmartUPS models) revealed an STM32F103 MCU, with an adjacent JTAG header.

PCB of SMT750I



PCB of SMT1500RM2UC

Through the JTAG interface we were able to dump the firmware in its unencrypted form, and analyze the bootloader and the firmware upgrade process.

This analysis showed that the firmware is encrypted with a symmetric encryption algorithm, and is decrypted during the firmware upgrade process, and stored unencrypted on the internal flash of the MCU.

Our primary device target, however, was the SMT1500RM2UC device - a popular model from APC's Smart-UPS line that supports the SmartConnect feature, in which the device connects to APC's Cloud, and is managed from the Internet.

Analysis of the PCB of the SMT1500RM2UC model revealed it has **two** MCUs, with two adjacent JTAG headers - a Texas Instruments' MCU (TM4C123B) and an STMicroelectronics MCU (STM32F217).

The Texas Instruments's MCU had an unlocked JTAG interface, and a quick analysis of its firmware revealed it is a secondary processor within the system, responsible for monitoring and controlling the power conversion logic of the board (more on that later).

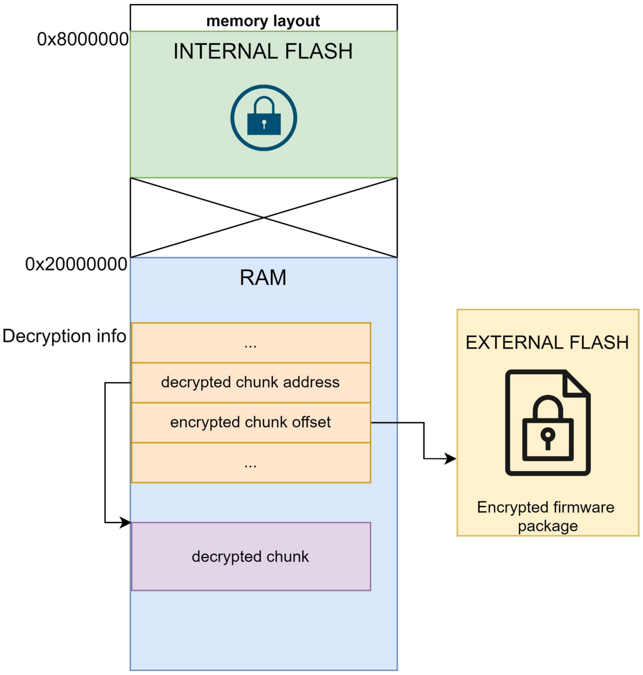
The main processor (the STM32), however, had a hardened JTAG interface, and the firmware could not be easily dumped through it.

The STM32 family of processors includes a Read Protection Unit (RDP) -- a hardware hardening feature meant to limit access through the JTAG interface (and other debug interfaces). It has three main operation modes:

- Level 0 - RDP protection is disabled.
- Level 1 - The debug interface is available, but accessing the content of the internal flash through it is disabled. The debug interface can be used to halt the and dump RAM memory -- but once the CPU is halted, execution can only be restored by hard-reset.
- Level 2 - The CPU can be halted through the JTAG interface, but access to all memory, including RAM is blocked.

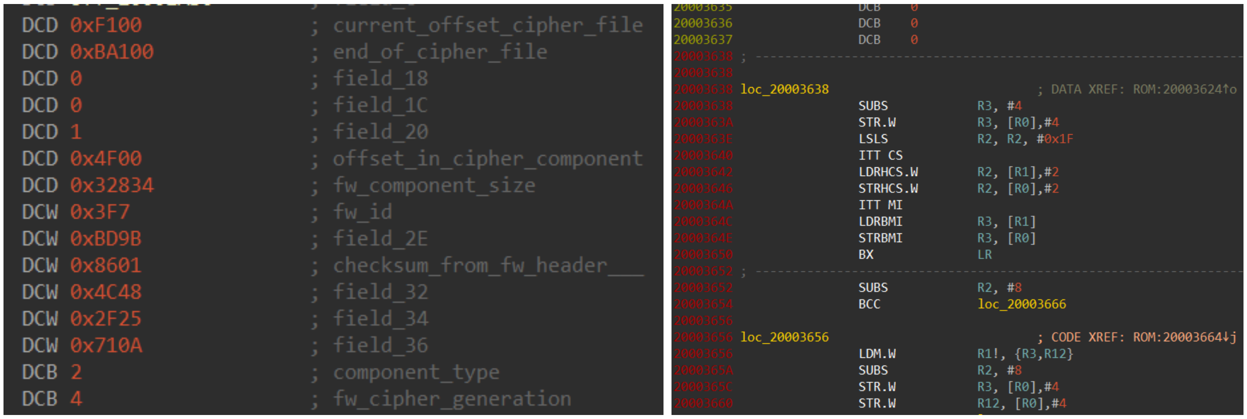
In the case of the STM32 processor in this Smart-UPS, RDP was set to level 1 -- so we were able to halt the CPU at a given time and dump the content of the RAM. But how can these capabilities help us recover the unencrypted firmware?

Bootloader analysis



By analyzing the bootloader of the SMT750I Smart-UPS, which didn't use any JTAG hardening methods, we found that during a firmware upgrade process, the encrypted firmware is decrypted in chunks. The unencrypted firmware is never loaded in full to the RAM -- the decryption process will iterate over all the encrypted chunks, and each iteration will decrypt a chunk to the RAM, store it in the flash, and continue to the next chunk. Since the same RAM memory is used for storing the decrypted chunks of firmware in each iteration, only one chunk of decrypted firmware data will be available in the RAM at a time.

Analyzing the decryption algorithm in the bootloader allowed us to learn the data structures of the decryption process, and use these to match a chunk of encrypted data from the upgrade file to a decrypted chunk in the RAM. While the SMT1500RM2UC seemed to have different memory mapping and decryption structures than the ones we've studied in the smaller UPS, similar magic constants within these structs were used to locate them in the newer version leading us to exfiltrate our first chunk of decrypted firmware code from the newer model as well:



```
DCD 0xF100                     ; current_offset_cipher_file
DCD 0xBA100                    ; end_of_cipher_file
DCD 0                          ; field_18
DCD 0                          ; field_1C
DCD 1                          ; field_20
DCD 0x4F00                     ; offset_in_cipher_component
DCD 0x32834                    ; fw_component_size
DCW 0x3F7                      ; fw_id
DCW 0xBD9B                     ; field_2E
DCW 0x8601                     ; checksum_from_fw_header___
DCW 0x4C48                     ; field_32
DCW 0x2F25                     ; field_34
DCW 0x710A                     ; field_36
DCB 2                          ; component_type
DCB 4                          ; fw_cipher_generation
```

```
20003635                    DCB     0
20003636                    DCB     0
20003637                    DCB     0
20003638 ; ---------------------------------------------------------------
20003638
20003638 loc_20003638                                  ; DATA XREF: ROM:20003624↑o
20003638                    SUBS            R3, #4
2000363A                    STR.W           R3, [R0],#4
2000363E                    LSLS            R2, R2, #0x1F
20003640                    ITT CS
20003642                    LDRHCS.W        R2, [R1],#2
20003646                    STRHCS.W        R2, [R0],#2
2000364A                    ITT MI
2000364C                    LDRBMI          R3, [R1]
2000364E                    STRBMI          R3, [R0]
20003650                    BX              LR
20003652 ; ---------------------------------------------------------------
20003652                    SUBS            R2, #8
20003654                    BCC             loc_20003666
20003656
20003656 loc_20003656                                  ; CODE XREF: ROM:20003664↓j
20003656                    LDM.W           R1!, {R3,R12}
2000365A                    SUBS            R2, #8
2000365C                    STR.W           R3, [R0],#4
20003660                    STR.W           R12, [R0],#4
```

Firmware decryption structures (left) and decrypted code (right).

Automated Extraction

A single decrypted chunk of firmware code would not be sufficient to start our vulnerability hunt, and accessing the entire unencrypted firmware required us to jump through additional hoops. While access to the JTAG interface allowed us to dump the content of the RAM memory at a chosen moment in time, it did not allow us to fully debug the MCU -- once the CPU was halted through the JTAG interface, the only way to resume execution is by hard-restarting the device. So accessing the entire decrypted firmware required building an automated setup that would perform the firmware upgrade process, halt the CPU at a certain time, dump the RAM, reboot the device - and then repeat the process in varying intervals, hoping to eventually catch a glimpse of all the decrypted chunks.

In the SMT1500RM2UC, the firmware upgrade process involves several steps:

1. The encrypted firmware is uploaded to the device (via USB, NMC or SmartConnect).
2. The encrypted firmware is stored to an external flash (while the internal flash stores the currently running firmware).
3. The firmware would now be either **installed** automatically (in certain conditions), or remain **pending**, and the user would then install it at a later time using the physical menu buttons on the UPS.
4. If the device senses it has sufficient battery power and is also connected to the AC power grid (redundancy of power for the following critical steps was probably in mind) - it will then reboot into a ‘recovery’ mode of the bootloader.
5. The bootloader will perform a verification process: decrypt the firmware, chunk-by-chunk, and roll the checksum of the decrypted firmware for later comparison.
6. Upon a successful verification, a decryption process will run **again,** decrypting the firmware chunk-by-chunk and storing each decrypted chunk in the internal flash.
7. A constant ‘magic’ will be stored in the new firmware, in the flash, indicating the switch-over between the old and the new firmware.
8. The device will reboot again, and the bootloader will run the new firmware.

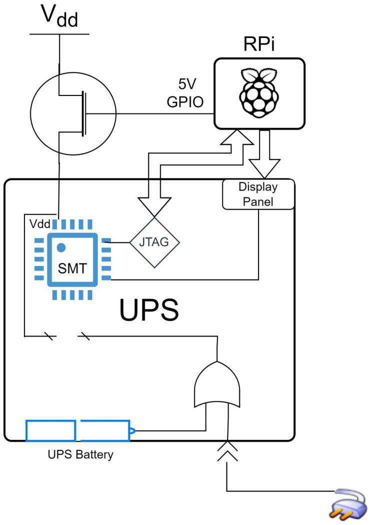
For our automated setup, steps 1-2 above were done manually, and the UPS was set to install the uploaded firmware via the physical buttons. The automated setup would then perform the following steps to extract the entire firmware:

1. Initiate the firmware install process (via physical buttons)
2. Sleep for a certain interval (while steps 4-5 above are performed - but before step 5 ends)
3. Halt the CPU via JTAG
4. Dump the RAM via JTAG
5. Reboot the device

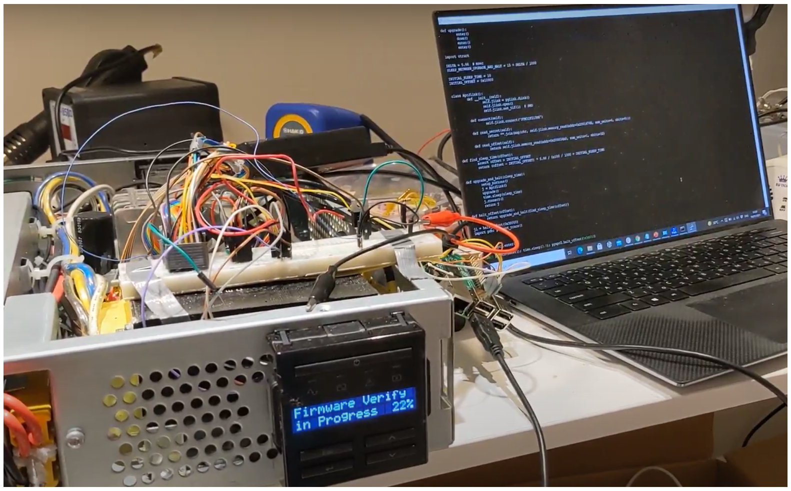
The above process will need to be repeated over and over again - increasing the sleep interval in each iteration a few milliseconds at a time. Building an automated setup that will perform the above steps posed two challenges:

1. Pressing the physical buttons to install the firmware would slow down the process considerably. **Solution:** the physical buttons were hooked up to GPIO inputs of an RPi, and controlled through it.
2. Rebooting the device once the CPU was halted required powering down the entire system, which had two power sources (battery + AC).
   Solution: The voltage regulator of the CPU itself was removed from the PCB, and the CPU power was then controlled by an external regulator with a switch, controlled by the RPi.

Finally, a Frankenstein-edition of the Smart-UPS was built in our lab:



Firmware extraction automation setup



A Smart-UPS being halted during ‘Firmware Verify’ stage

The RPi orchestrated the entire process of the firmware extraction - controlling the physical buttons in the menu of the Smart-UPS through GPIOs, halting the CPU and dumping the RAM via the JTAG interface, and rebooting the device by controlling a FET connected to the voltage regulator of the CPU. Running this setup repeatedly for a day or so allowed us to extract the entire unencrypted content of the firmware.

Discovered vulnerabilities

Firmware signing vulnerability - CVE-2022-0715

Our quest to extract an unencrypted firmware from a Cloud-connected Smart-UPS has led us to the following conclusions:

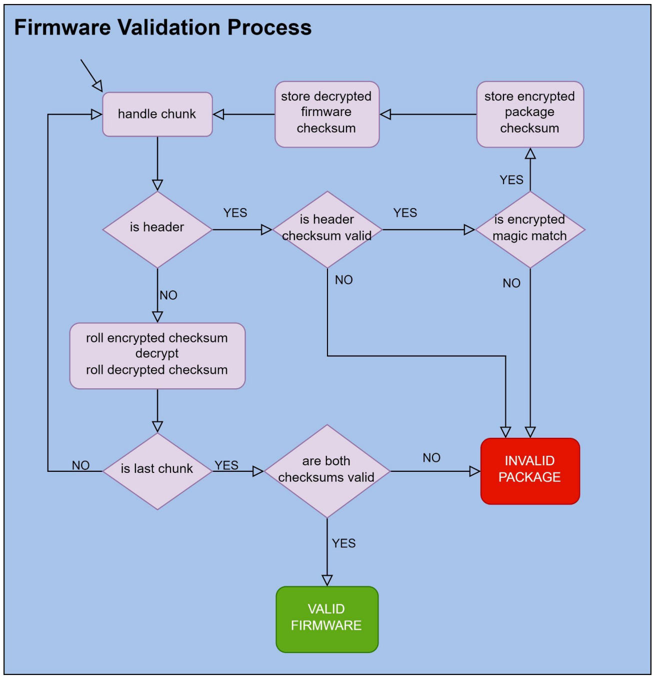
- Smart-UPS firmware files are encrypted using symmetric encryption.
- Each sub-brand in the Smart-UPS line uses a different encryption key, but the key is identical to all units of a particular sub-brand.
- Since the encryption key is symmetrical, it is present on the internal flash of Smart-UPS devices (each unit contains only the encryption keys for its sub-brand).

- By harvesting an encryption key from a single unit of a certain sub-brand (using techniques such as ones we've demonstrated in the previous section, or by abusing a known vulnerability in an unpatched device), it is possible to decrypt all firmware files of a certain sub-brand.

An additional notion had come to mind - does APC rely on the symmetrical encryption as a form to sign firmware files? The final stage of the bootloader's decryption algorithm calculates the checksum of the decrypted firmware chunks. If an attacker attempts to upgrade the firmware of a target Smart-UPS with a binary blob that was not encrypted using the correct encryption keys, this last validation step would prevent him from installing an unwanted firmware.

Despite this - symmetrical encryption + a checksum validation are not a cryptographically secure method of code signing, especially when the encryption keys are identical across all units of a certain sub-brand of Smart-UPS devices. Once an encryption key is exfiltrated from a single unit, malicious firmware files can then be produced and used to target any other unit from the same family.

Further analysis of the Smart-UPS bootloader revealed that in fact, no additional firmware signing methods are in use. Detailed below is the entire firmware validation process performed by APC:



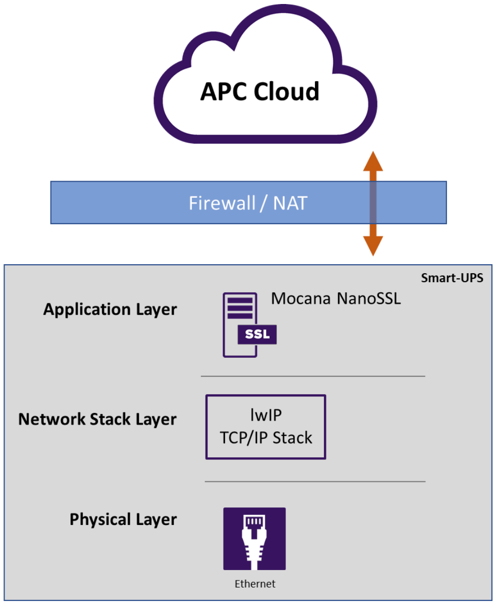
Simplified flow of firmware package validation

Overall the firmware performs four validation steps that check the validity of four values passed in the firmware header. These steps validate three different checksums, and one constant magic stored within the encrypted portion of the firmware. While an intimate knowledge of the bootloader validation steps is required to properly construct a valid firmware, the lack of any cryptographic signature measures leaves the validation process with a fatal integrity flaw. Having extracted the encryption keys from two different models (SMT750I and SMT1500RM2UC) and analyzing the firmware's structure and encryption algorithms, we were able to forge malicious firmware packages and install them on target UPS devices.

As detailed in the background section, Smart-UPS devices support multiple firmware upgrade paths - either physically (through USB) or through the network (via the NMC card, or via the Cloud-connection). An attacker that is able to forge malicious firmware files can abuse one of these paths to gain code-execution on target devices, and abuse it either for altering the proper operation of the UPS (see the 'Impact' section below), or using the UPS as a gateway to reach internal networks and as a stronghold within them.

While the firmware upgrade process via USB requires physical access to the device, the network-based firmware upgrade paths can be done remotely. Moreover, the NMC allows an upgrade to be initiated through its internal web server that by default uses either HTTP or HTTPS (depends on NMC model and configuration) with a self-signed certificate. This means that an attacker can perform a man-in-the-middle attack against the UPS (via DNS spoofing or ARP spoofing, for example), and gain access to the NMC's password. Using this password he can then install a malicious firmware through the NMC's web server and gain remote-code-execution via an unauthenticated attack.

Going down the TLS rabbit hole



As detailed in the firmware extraction section, our main target of interest was the SMT1500RM2UC model, one of the next-generation Smart-UPS models that support the SmartConnect feature. Having gained access to the device's unencrypted firmware, we were able to start mapping the attack surface exposed by its connection to the Cloud.

The device uses a bare-metal OS (no OS), while utilizing the open-source lwIP TCP/IP stack as a primary networking stack, and the 3rd party closed-source Mocana NanoSSL library as its primary TLS library.

If an NMC card is present on a Smart-UPS, the network attack surface the device exposes may contain the internal web server of the NMC card, as well as SNMP client\server, Modbus over TCP, and potentially other local network services it supports.

However, when the SmartConnect feature is in use, the device's network exposure is reduced dramatically -- since the device will only use the network to acquire an IP address (DHCP client, or static configuration), query the DNS server (DNS client), and connect to the APC Cloud using TLS over TCP. Thus the primary attack surface is thus lwIP itself - a lightweight open-source and heavily researched networking stack, and Mocana's NanoSSL - a closed-source proprietary TLS library.

Mocana is a startup company, recently acquired by DigiCert, that offers various solutions for securing embedded industrial and IoT devices, as well as supplying highly reliable, FIPS-certified, Crypto, TLS and SSH libraries. Amongst its leading customers, you may find large OT companies, such as ABB, Siemens, Schneider Electric, and GE, as well general IoT companies such as Xerox. Despite the popularity, and sensitivity of Mocana's embedded TLS library, no public research projects have been published as of yet that focus on its security.

When constructing an application that operates over TLS and TCP, an additional 'glue-logic' layer is needed to connect the dots - in this case, a glue-logic layer is required to connect the application running on top of TLS (CoAP in this instance - more on that, later) with the TLS library (nanoSSL), and with the TCP library (lwIP). As we'll see below, this glue-logic layer can be an unexpected source of edge case bugs, when handling specific sensitive error flows.

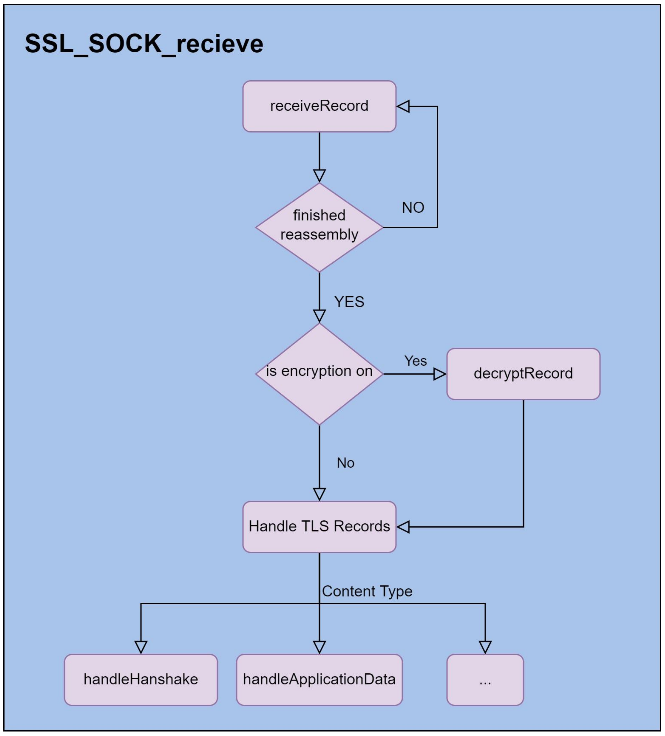
TLS Packet Reassembly Vulnerability - CVE-2022-22805

Since TLS usually operates over frameless protocols such as TCP, it includes a basic header with a length field used to reconstruct TLS records sent from peer to peer. TLS implementations will first attempt to receive this basic header information, extract the length of the expected TLS record from it, and then reassemble the TLS record from the incoming TCP stream. The basic TLS header contains only three fields:

Byte offset	0	1	2	3	4
Field	Content-Type	Version		Length	

TLS Header

These fields are always non-encrypted, since they are part of the basic framing mechanism of TLS, utilized by TLS records used both in the TLS handshake phase (pre-authentication), and in the TLS application data phase. In an attempt to map the pre-authentication attack surface of the nanoSSL TLS library, we honed in on the code flow that implements this framing layer in TLS:



Simplified code flow of SSL_SOCK_receive function in nanoSSL

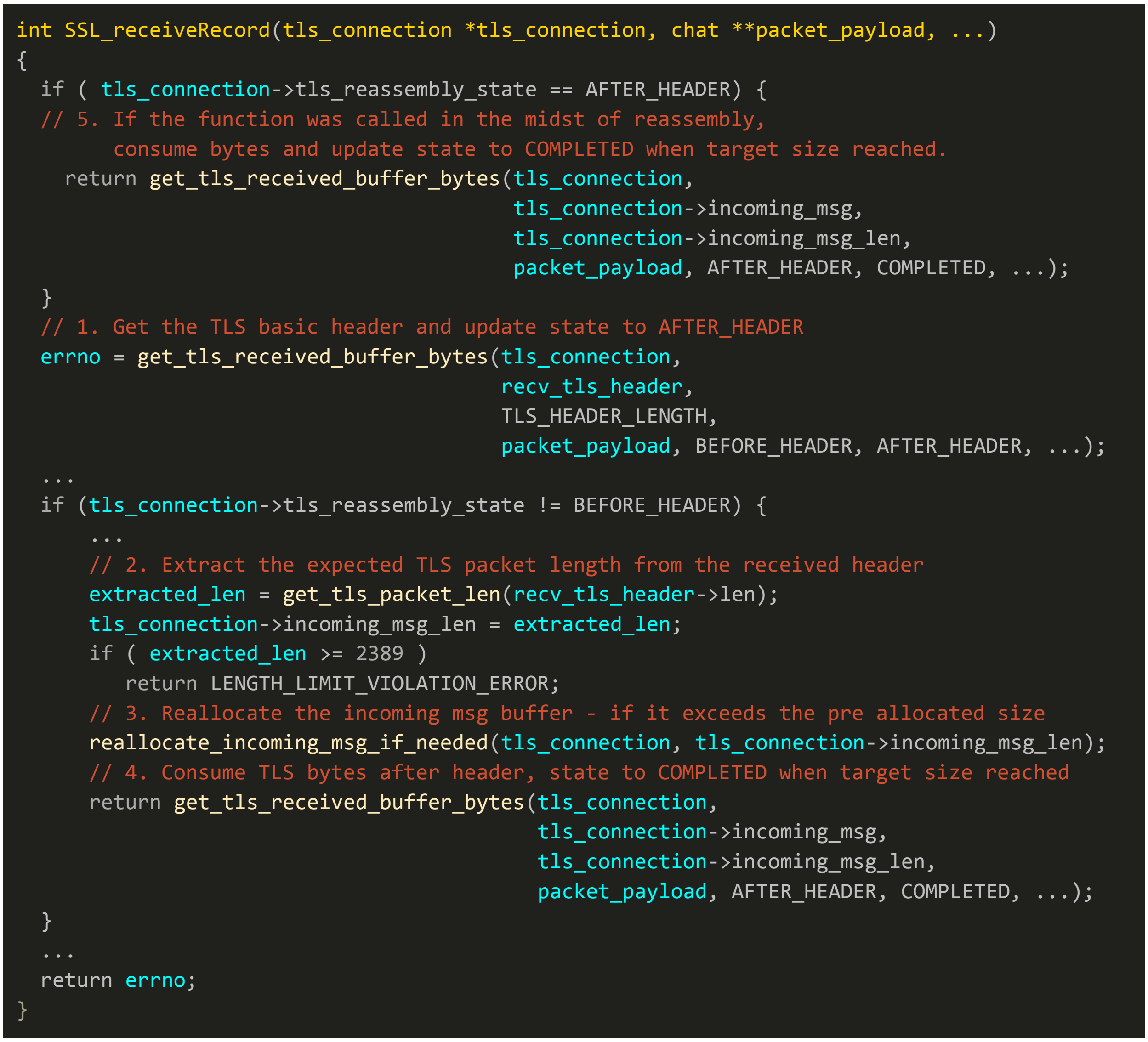
The *SSL_SOCK_receive* function is called by the TLS library whenever incoming bytes are received from the underlying transport layer (TCP, or UDP in the case of DTLS), and can be triggered either by synchronous or asynchronous API functions of nanoSSL (*SSL_receive* and *SSL_ASYNC_recvMessage*, respectively). The *receiveRecord* function unpacks the basic TLS header, and reassembles incoming TLS records. It maintains the reassembly state, and will continue to aggregate the incoming bytes until a complete TLS record is received. Then, the function will handle the received record - first, decrypt it (if encryption had already been enabled in the TLS connection - after the TLS handshake), and then handle the various TLS messages (either ones relating to the establishment of the connection, during the TLS handshake, or ones that may occur throughout the session, such as the TLS Alert message). Lastly, if the incoming TLS message is a TLS application data message, the TLS headers will be stripped, and the payload will be returned to the upper-layer application.

Whenever an internal error condition occurs in either of these steps, it will be returned to the caller of this function -- which normally would be handled by the glue-logic layer, connecting the TLS library with the specific application in use, and the TCP/IP stack.

As noted above, the *SSL_receiveRecord* function reassembles the incoming TLS records and maintains a reassembly state:

- ***BEFORE_HEADER*** - The initial state before reassembly began.
- ***AFTER_HEADER*** - the state after the basic TLS header was extracted and parsed, and before the TLS record was fully reassembled (based on the length specified in the header)
- ***COMPLETED*** - the final state after the entire TLS record was reassembled.

A simplified version of the decompiled reassembly function can be seen here:



```
int SSL_receiveRecord(tls_connection *tls_connection, chat **packet_payload, ...)
{
  if ( tls_connection->tls_reassembly_state == AFTER_HEADER) {
  // 5. If the function was called in the midst of reassembly,
        consume bytes and update state to COMPLETED when target size reached.
    return get_tls_received_buffer_bytes(tls_connection,
                                         tls_connection->incoming_msg,
                                         tls_connection->incoming_msg_len,
                                         packet_payload, AFTER_HEADER, COMPLETED, ...);
  }
  // 1. Get the TLS basic header and update state to AFTER_HEADER
  errno = get_tls_received_buffer_bytes(tls_connection,
                                        recv_tls_header,
                                        TLS_HEADER_LENGTH,
                                        packet_payload, BEFORE_HEADER, AFTER_HEADER, ...);
  ...
  if (tls_connection->tls_reassembly_state != BEFORE_HEADER) {
      ...
      // 2. Extract the expected TLS packet length from the received header
      extracted_len = get_tls_packet_len(recv_tls_header->len);
      tls_connection->incoming_msg_len = extracted_len;
      if ( extracted_len >= 2389 )
         return LENGTH_LIMIT_VIOLATION_ERROR;
      // 3. Reallocate the incoming msg buffer - if it exceeds the pre allocated size
      reallocate_incoming_msg_if_needed(tls_connection, tls_connection->incoming_msg_len);
      // 4. Consume TLS bytes after header, state to COMPLETED when target size reached
      return get_tls_received_buffer_bytes(tls_connection,
                                           tls_connection->incoming_msg,
                                           tls_connection->incoming_msg_len,
                                           packet_payload, AFTER_HEADER, COMPLETED, ...);
  }
  ...
  return errno;
}
```

Simplified decompilation of *SSL_receiveRecord*

The reassembly state is held in the *tls_reassembly_state*, and *incoming_msg* holds the received bytes (*packet_payload*), until the TLS record is fully reassembled. The *incoming_msg* pointer in the TLS connection object is first pre-allocated in the heap, when the TLS socket is initiated, and the *incoming_msg_len* holds the length of the TLS record.

The *get_tls_received_buffer_bytes* function is responsible for copying the input bytes to the reassembled buffer at the correct reassemble offsets, while validating they don't exceed the allocated buffer size. It will also advance the *packet_payload* pointer by the number of bytes it had consumed, and advance the reassembly state when the expected bytes have been accumulated (it receives the current state, as well as the next reassembly state, and will both validate and advance the state when needed).

The following reassembly steps are achieved within the function above (the step numbers are marked in code comments):

1) Receive the header. This updates the state from **BEFORE_HEADER** to **AFTER_HEADER**.
2) Parse the length received from the header and update *incoming_msg_len* (which holds the expected length of the reassembled TLS record).
3) Reallocate the dedicated buffer (*incoming_msg*), if the required length exceeds its initial length of the allocated buffer.
4) Consume the remaining bytes (in *packet_payload*) to the reassembled TLS record (*incoming_msg*). If the entire record has been consumed, the state will update to **COMPLETED**, and if not it will remain **AFTER_HEADER**.
5) When additional bytes are received in *SSL_SOCK_*receive, the *receiveRecord* function will be called again, and the incoming bytes will be consumed by called *get_tls_received_buffer_bytes* until the expected length is received, and state will update to **COMPLETED**.

Once the reassembly state changes to **COMPLETED**, parsing and handling of the incoming TLS record will continue in *SSL_SOCK_receive*.

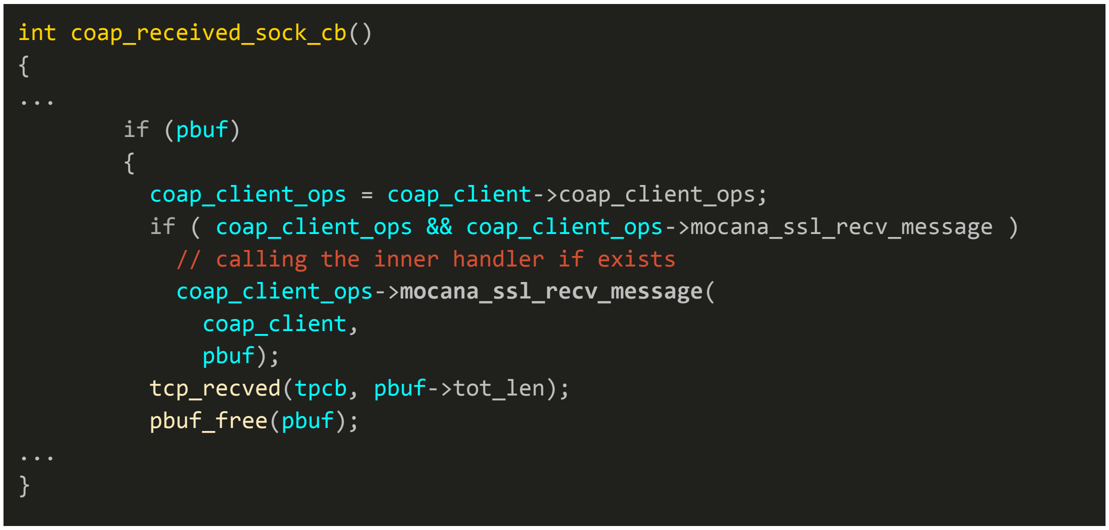
While the reassembly process above is a bit complex, it does attempt to protect the reassembly buffer (*incoming_msg*) from potential overflows. Despite this, taking a deeper look in the *receiveRecord* function reveals an interesting edge case. Between steps 2 and 3, a size check takes place - if the extracted length exceeds a certain constant (2389), the reassembly process stops and returns an indicative error number, but the reassembly variables in the TLS connection object are left in an in-between state:

1) The reassembly state is **AFTER_HEADER**
2) The *incoming_msg_len* is the length from the extracted TLS record (that generated the error condition)
3) The *incoming_msg* buffer has not been reallocated using the *incoming_msg_len* (the function returned before this occurred), so it is pointing to the initial buffer allocated when the socket was initiated (by default, a 340-byte buffer)

As noted above, when an error condition occurs within *SSL_SOCK_receive*, it will be returned to the application, through the API functions. In the edge case detailed above (and in additional ones, we'll show below), it is apparent that the nanoSSL library regards these error conditions as ones that require the termination of the TLS connection, and a cleanup of the TLS connection object, and rely on the application to call the API function *SSL_close*, to achieve these goals.

In the above edge case, failure to close the TLS connection can lead to a pre-authentication heap overflow condition that can lead to remote-code-execution. This is a result of the reassembly state being left in the in-between state shown above. After this state is achieved, and given that the TLS connection was not properly closed, processing of additional bytes through *SSL_SOCK_receive* will lead to the internal function *get_tls_received_buffer_bytes* to overflow the *incoming_msg* buffer, allocated on the heap.

As is probably clear at this point, the glue-logic implemented in the Smart-UPS doesn't handle the returned error codes from the nanoSSL API function that handles the SSL socket received data:



```
int coap_received_sock_cb()
{
...
        if (pbuf)
        {
          coap_client_ops = coap_client->coap_client_ops;
          if ( coap_client_ops && coap_client_ops->mocana_ssl_recv_message )
            // calling the inner handler if exists
            coap_client_ops->mocana_ssl_recv_message(
              coap_client,
              pbuf);
          tcp_recved(tpcb, pbuf->tot_len);
          pbuf_free(pbuf);
...
}
```

Simplified decompilation of *coap_received_sock_cb*

The CoAP (Constrained Application Protocol) glue-logic above calls a wrapper to nanoSSL's *SSL_receive* function (*mocana_ssl_recv_message*), which propagates the internal error code - which is ultimately dropped by the function above, and unhandled.

To trigger the above heap overflow, two simple steps are required:

1. A TLS header is sent (pre-authentication) to a target device, in which the *length* field is specified as a large number (over 2389 bytes). This leads to an unhandled error condition in which the reassembly context is partially preserved - the expected packet size extracted from the dropped packet is set to be used in the next packet reception process, although a memory with the matching size wasn't allocated.
2. Any packets received at this state in the TLS socket will be handled as if reassembly is still in progress, accumulating the incoming bytes in the *incoming_message* buffer, until the size specified in the previous step is reached.

Exploitability

The size specified in the first TLS header is fully controlled by the attacker, and can be chosen to be larger than the allocation size of the *incoming_msg* (340 bytes), leading the reassembly process to result in a heap overflow with attacker-controlled data.

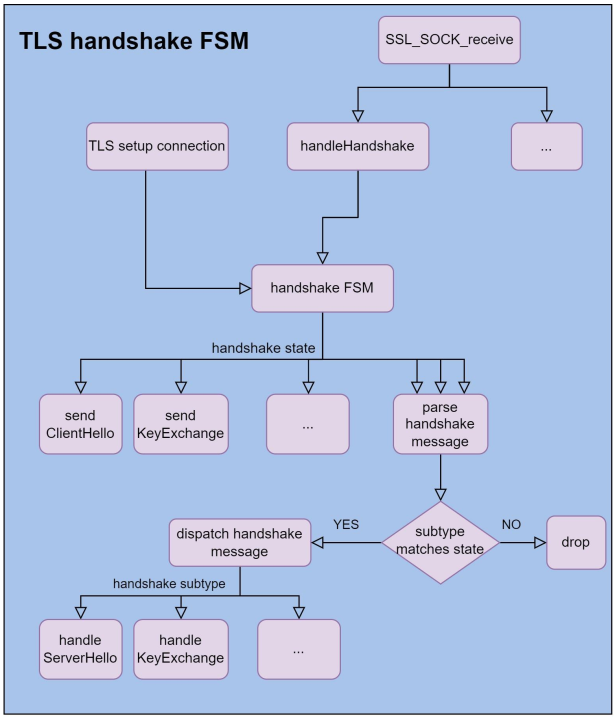
This critical vulnerability can be used to attack both vulnerable TLS clients and servers, and can be initiated at any TLS connection state - either pre-authentication (before\during the TLS handshake) or post-authentication (after encryption had been turned on).

The Smart-UPS devices we've researched don't utilize any exploit mitigation mechanisms, or any heap sanitization techniques that might complicate exploitation of this vulnerability. Thus reaching remote-code-execution with this vulnerability can be done in a reliable way, as we demo in a video on our blog. Further detail of the exploit development of this vulnerability will be provided in a future write-up.

TLS Handshake Authentication Bypass - CVE-2022-22806

The most straightforward pre-authentication attack surface of TLS is in the parsing and handling of messages in the handshake phase - since it occurs before the connection is encrypted. The vulnerability in the previous section detailed a certain class of bugs - improper handling of error conditions, and we decided to explore whether additional paths of error handling (such as those in the processing of TLS handshake messages) were similarly flawed.

First, we wanted to have a better understanding of the state machine of the TLS handshake process. It is managed by one main function (*handleHandshake*), and called by the aforementioned *SSL_SOCK_receive*:



Simplified code flow of the TLS handshake finite state machine

The TLS connection object holds a state variable which navigates the progression flow of each handshake step. Each handler is responsible for implementing its step according to the RFC, and updating the TLS connection object variables (including the state variable, when needed). The first message that a TLS client (such as the SmartConnect TLS client) expects to receive, after it sends the first ClientHello message, is the ServerHello message. A simplified version of the decompiled ServerHello handler function can be seen here:

```
int parse_server_hello_message(tls_connection_t *tls_connection, char *server_hello_message)
{
...
   // Handle Session ID
   if (tls_connection->session_id_length > 0 &&
       memcmp(received_session_id, tls_connection->session_id, tls_connection->session_id_length)) {
     // Setting tls_resumption boolean if the received session ID matched the cached value
     tls_connection->tls_resumption = TRUE;
   } else {
     // New TLS session - setting the connection’s Session ID
     tls_connection->tls_resumption = FALSE;
     tls_connection->session_id_length = session_id_length;
     memcpy(tls_connection->session_id, session_pointer, session_id_length);
   }
...
   // Validating the chosen cipher suite
   if (matched_cipher_suite = match_cipher_suite(tls_connection) && matched_cipher_suite)
     tls_connection->cipher_suite = matched_cipher_suite;
   else
     return CIPHER_SUITE_MISMATCH;
   if (tls_connection->tls_resumption) {
     // Copying the master secret from the cache and
     // progressing the handshake state to post key exchange
     memcpy(tls_connection->master_secret, tls_connection->cached_master_secret, 48);
     tls_connection->tls_handshake_state = AFTER_KEY_EXCHANGE;
   }
   else
   {
     // progress the handshake state to the next one
     tls_connection->tls_handshake_state = PEDNING_KEY_EXCHANGE;
   }
...
}
```

Simplified decompilation of *parse_server_hello_message*

The function first handles the session ID field -- if the supplied session ID matches the session ID saved in the current TLS connection object it will assume the TLS resumption feature is in use (see the Background section), and set a matching field in the connection object to indicate this state. If not, the connection’s session ID will be taken from the incoming message. The function will then handle the cipher suite offered by the server in the ServerHello message. In a nominal TLS handshake, the client first offers a set of cipher suites it supports, and the server will choose a cipher suite from the list and offer it in the ServerHello message.

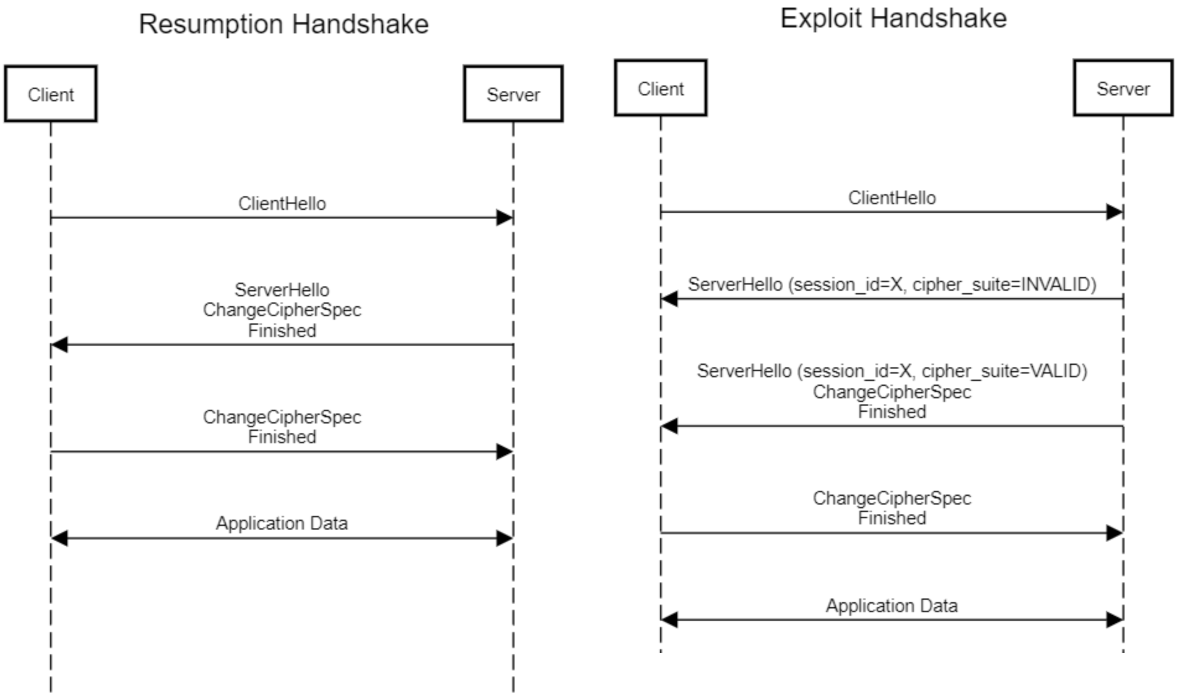
An interesting edge case can be reached if the ServerHello message consists of a cipher suite the client doesn’t support (and was therefore not offered by him in the ClientHello message). In this case, the function will fail and return errno *CIPHER_SUITE_MISMATCH*, without updating the *tls_handshake_state*. The function assumes this errno will be treated by the upper layer by terminating the TLS connection. The errno is passed through the calling functions to *SSL_SOCK_receive* but in the case of *SmartConnect* TLS client, it is ultimately ignored in the *coap_received_sock_cb* function.

If the TLS connection is not terminated, the TLS connection's state is left in an unexpected state -- in which additional ServerHello messages may be handled by the TLS library. A second ServerHello that is parsed in this state may abuse the fact that the session ID variables (*session_id_length* and *session_id*) have already been set while handling the first ServerHello message. This can lead the function to assume a successful TLS resumption condition has been reached although a successful handshake hadn't actually occurred.

This can allow an attacker to exploit this state confusion to completely bypass the authentication phase by sending two consecutive ServerHello messages:

1) The first message will include a cipher suite the client doesn't support (this can be a faulty cipher suite ID), and a randomly chosen session ID.
2) The second message will include a valid cipher suite and the same session ID from the previous ServerHello packet.

This will lead to a flow where the session is treated as a TLS resumption session, although the Master Secret hasn't been generated yet, resulting in it being fetched from an array within the TLS connection object that was initialized to zeros upon creation. Thus, processing of the second ServerHello message in this state will lead the Master Secret of the connection to be 48 bytes of zeros - a fixed and predictable constant which can be used by the attacker to successfully finish the TLS resumption setup, and move on to the application phase uninterruptedly as though he were a trusted server.



Resumption TLS handshake vs Exploit TLS handshake

Exploitability

The authentication bypass vulnerability detailed above undermines basic security assumptions used by APC in the design of the SmartConnect feature. Similar applications of Cloud-connected devices would suffer from a similar fate if their usage of the nanoSSL library improperly handles the returned error codes.

If an attacker is able to establish a man-in-the-middle position - either within a local network or beyond the perimeter directly from the Internet, they can exploit this vulnerability to masquerade as the APC Cloud. This would allow them control over the connected Smart-UPS device through any of the available functionality that APC has implemented in the SmartConnect feature. This includes the option to perform remote firmware upgrades (as detailed in the Background section), without any user interaction. Combining this vulnerability with the firmware signing vulnerability (CVE-2022-0715) can allow an attacker to gain remote-code-execution on target devices, directly from the Internet. As a single firmware package includes the code for all updatable units of the system, this vulnerability can allow an attacker to control all components of the UPS - that has multiple processors with different roles (detailed in the next section).

Impact

While a theoretical compromise of a Cloud-Connected UPS sounds ominous, we wanted to explore the real-world potential of such an attack, to better assess its actual risk.

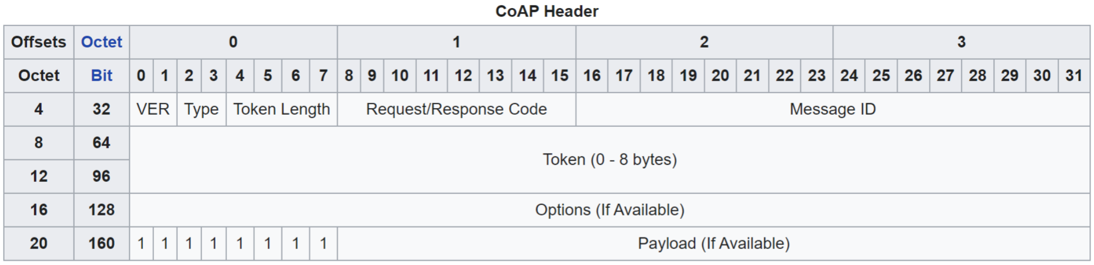
APC Cloud interface

The most reliable path for attackers to gain code-execution by exploiting one of these vulnerabilities is to exploit the TLS Authentication Bypass vulnerability presented above. To do so, they would first be required to establish a man-in-the-middle position on the target device, and while this is not trivial, various attacks in the past have demonstrated such capabilities, for example, via DNS hijacking. Having established such a position, the attacker can exploit the authentication bypass vulnerability, as detailed above, and control Smart-UPS devices that connect to it via the APC Cloud interface.

This Cloud interface is based on a well-known protocol - LwM2M. This protocol (which stands for Lightweight-Machine-to-Machine, specified by OMA SpecWorks) is quite common in various Cloud-connected device applications. The underlying protocol of LwM2M is CoAP (Constrained Application Protocol).

CoAP

The CoAP protocol is defined by its RFC as a “specialized web transfer protocol for use with constrained nodes and constrained networks”. In many ways, it resembles a lightweight and binary compact version of HTTP, with request\response commands, and flexible structures that can facilitate transfer of messages in varying formats. Unlike HTTP, it is a binary protocol in which both peers can initiate a request\response transaction, but similarly to HTTP, it uses URIs and request methods, similar to HTTP’s GET\POST requests.

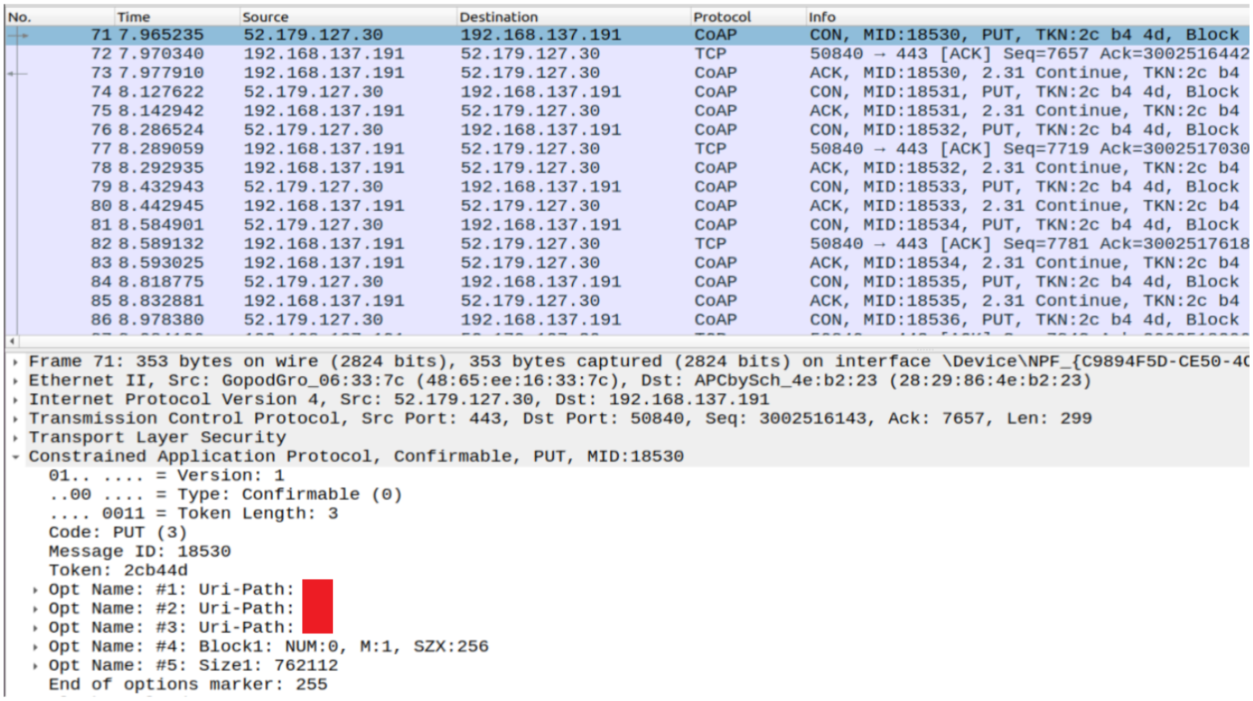


<table>
<caption>CoAP Header</caption>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>4</td><td>32</td><td colspan="2">VER</td><td colspan="2">Type</td><td colspan="4">Token Length</td><td colspan="8">Request/Response Code</td><td colspan="16">Message ID</td></tr>
<tr><td>8</td><td>64</td><td colspan="32" rowspan="2">Token (0 - 8 bytes)</td></tr>
<tr><td>12</td><td>96</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Options (If Available)</td></tr>
<tr><td>20</td><td>160</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="24">Payload (If Available)</td></tr>
</table>

CoAP header, from wikipedia

LwM2M

The LwM2M protocol defines a specific applicative usage of CoAP. it standardizes CoAP URIs for various interfaces, such as firmware update, device information, etc. It fully details how to use CoAP to access each of the common interfaces, while reserving additional URIs for OEM proprietary use cases. A standard usage of LwM2M includes a client registration request and a server response, followed by a series of server requests and client replies. A simple LwM2M interaction can be described by a client that registers to a server that manages it by sending commands to relevant paths (URIs).For example, there is a path dedicated to sending firmware update blocks, as shown below from decrypted TLS session in which firmware update blocks are sent to a SmartUPS by the APC Cloud:



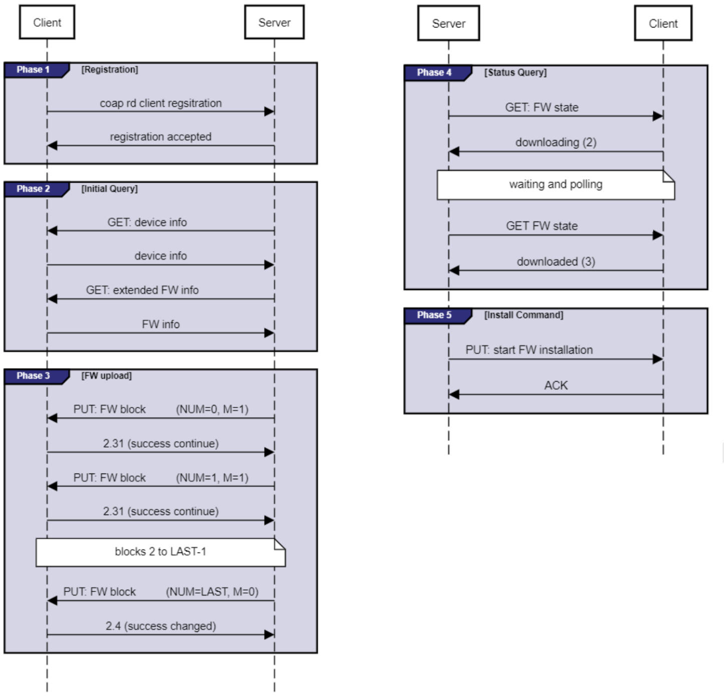
```
No.   Time        Source            Destination       Protocol  Info
   71 7.965235    52.179.127.30     192.168.137.191   CoAP      CON, MID:18530, PUT, TKN:2c b4 4d, Block
   72 7.970340    192.168.137.191   52.179.127.30     TCP       50840 → 443 [ACK] Seq=7657 Ack=3002516442
   73 7.977910    192.168.137.191   52.179.127.30     CoAP      ACK, MID:18530, 2.31 Continue, TKN:2c b4
   74 8.127622    52.179.127.30     192.168.137.191   CoAP      CON, MID:18531, PUT, TKN:2c b4 4d, Block
   75 8.142942    192.168.137.191   52.179.127.30     CoAP      ACK, MID:18531, 2.31 Continue, TKN:2c b4
   76 8.286524    52.179.127.30     192.168.137.191   CoAP      CON, MID:18532, PUT, TKN:2c b4 4d, Block
   77 8.289059    192.168.137.191   52.179.127.30     TCP       50840 → 443 [ACK] Seq=7719 Ack=3002517030
   78 8.292935    192.168.137.191   52.179.127.30     CoAP      ACK, MID:18532, 2.31 Continue, TKN:2c b4
   79 8.432943    52.179.127.30     192.168.137.191   CoAP      CON, MID:18533, PUT, TKN:2c b4 4d, Block
   80 8.442945    192.168.137.191   52.179.127.30     CoAP      ACK, MID:18533, 2.31 Continue, TKN:2c b4
   81 8.584901    52.179.127.30     192.168.137.191   CoAP      CON, MID:18534, PUT, TKN:2c b4 4d, Block
   82 8.589132    192.168.137.191   52.179.127.30     TCP       50840 → 443 [ACK] Seq=7781 Ack=3002517618
   83 8.593025    192.168.137.191   52.179.127.30     CoAP      ACK, MID:18534, 2.31 Continue, TKN:2c b4
   84 8.818775    52.179.127.30     192.168.137.191   CoAP      CON, MID:18535, PUT, TKN:2c b4 4d, Block
   85 8.832881    192.168.137.191   52.179.127.30     CoAP      ACK, MID:18535, 2.31 Continue, TKN:2c b4
   86 8.978380    52.179.127.30     192.168.137.191   CoAP      CON, MID:18536, PUT, TKN:2c b4 4d, Block

› Frame 71: 353 bytes on wire (2824 bits), 353 bytes captured (2824 bits) on interface \Device\NPF_{C9894F5D-CE50-4C
› Ethernet II, Src: GopodGro_06:33:7c (48:65:ee:16:33:7c), Dst: APCbySch_4e:b2:23 (28:29:86:4e:b2:23)
› Internet Protocol Version 4, Src: 52.179.127.30, Dst: 192.168.137.191
› Transmission Control Protocol, Src Port: 443, Dst Port: 50840, Seq: 3002516143, Ack: 7657, Len: 299
› Transport Layer Security
⌄ Constrained Application Protocol, Confirmable, PUT, MID:18530
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0011 = Token Length: 3
    Code: PUT (3)
    Message ID: 18530
    Token: 2cb44d
  › Opt Name: #1: Uri-Path: [redacted]
  › Opt Name: #2: Uri-Path:
  › Opt Name: #3: Uri-Path:
  › Opt Name: #4: Block1: NUM:0, M:1, SZX:256
  › Opt Name: #5: Size1: 762112
    End of options marker: 255
```

Decrypted LwM2M stream of firmware download as sniffed with Wireshark.

APC Firmware upgrade over LwM2M

The APC firmware upgrade process uses several interfaces:

After approval of the client registration request, the server queries the device information object and determines whether a firmware upgrade process is needed. If a firmware upgrade is required, it will use the firmware update object to consecutively send the firmware blocks, and initiate the firmware installation using an APC proprietary object.



APC Firmware upgrade flow in LwM2M

Having reversed-engineered the APC firmware upgrade protocol we were able to take over a Smart-UPS from the Internet, using the following steps:

1. Establish a man-in-the-middle position to target a specific network (our own network, but from the Internet).
2. Hijack the APC Cloud DNS URL and point it to our 'attacker' server on the Internet.
3. Once the Smart-UPS within the internal network reaches out to our 'attacker' server through TLS, exploit the TLS authentication bypass vulnerability (CVE-2022-22806), leading the UPS to trust our server as if it were the true APC Cloud service.
4. Using the exploited TLS connection, abuse the APC firmware upgrade protocol (over LwM2M) and install a malicious firmware on the Smart-UPS.
5. This malicious firmware would include a RAT (remote access trojan), that would connect to our 'attacker' server, and allow us to fully control the UPS:
    a. Read\Write memory
    b. Execute our own code that can abuse the functionality of the UPS (more on that, below)

c. Use the UPS as a gateway, to interact with the internal network, for launching additional attacks

Abusing a UPS

A benign-looking UPS may actually store significant potential for hazardous attacks, which can be remotely executed by abusing vulnerabilities such as the ones detailed above. A UPS is essentially a small-scale power production facility, encompassed in a compact box. It holds a large DC battery that stores energy, and regulates the charging voltage, speed, and current while monitoring various environmental conditions to avoid damaging the battery. When a power shortage occurs it will invert the battery's DC energy to AC, while regulating the AC voltage in a closed-loop circuit based on the actual power consumed by the downstream devices that rely on the UPS.

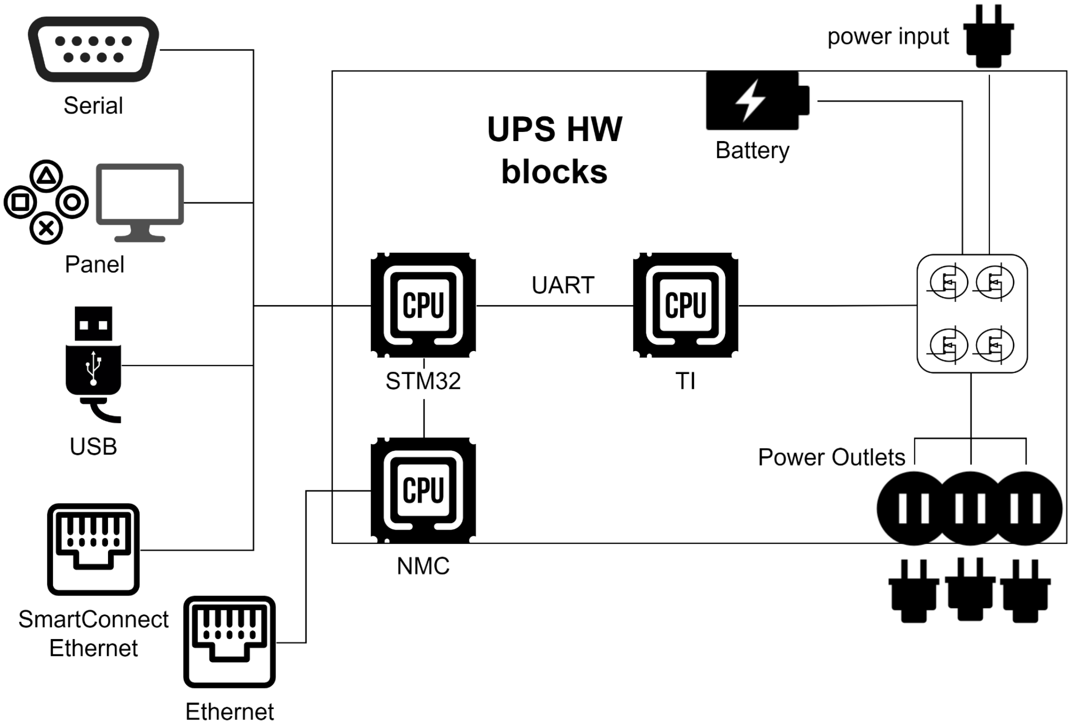
Thus, the potentially hazardous effects of abusing the software of a UPS may include the following scenarios:

1. The outputs of the UPS may be turned off - simply to disrupt the power to downstream devices that rely on it.
2. The outputs of the UPS may rapidly change from on to off, which can damage sensitive downstream devices.
3. The output voltage and the AC frequency can be dramatically altered - which can again, damage sensitive downstream devices.
4. The battery charging parameters may be severely altered, which can damage the battery, or in extreme cases potentially lead the battery to overheat.
5. Various parameters of the DC to AC conversion mechanism can be severely altered, which can lead the UPS itself to overheat - and potentially smoke, and eventually - **self-destruct**.

To explore the real-life extent of the potential effects listed above, we chose to dive deep into the inner workings of the Smart-UPS and attempt to create some of these effects in our lab.

UPS hardware design

The SmartUPS is based on two separate processors (MCUs) - A main processor - an **STM32** MCU, responsible for connecting the UPS to external communication interfaces, and a secondary processor - a Texas Instruments (**TI**) MCU, responsible for power regulation. The two MCUs communicate with each other using a UART interface.



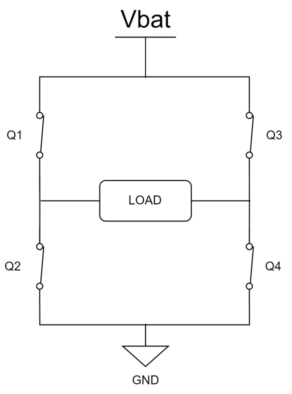
UPS hardware design main blocks. The MOSFETs formation connecting the TI CPU to the outlets represents the electrical circuit “black box”.

Power regulation

As mentioned above, the Texas Instruments processor in the design of the Smart-UPS is tasked with regulating the power conversion mechanism of the UPS. The mechanism is responsible for converting AC to DC power (when AC is connected), charging the battery, and switching over to DC to AC power conversion when AC power is not available. While the AC to DC conversion is relatively simple to implement, the other direction requires multiple signal processing techniques. The complexity of these techniques is dependent on the required characteristics of the reconstructed output sine signal.

H-bridge

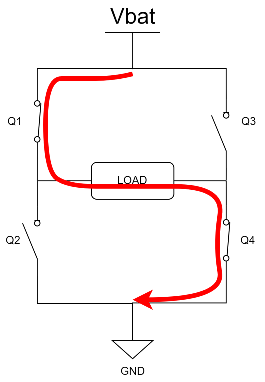
The most naive implementation for inverting a DC signal to an AC one is by simply changing the polarity of the DC current at equally timed intervals. This naive implementation would result in an AC signal of a square wave, and can be done using a formation of switches called an H-bridge.



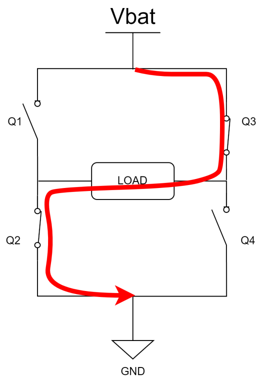
H-bridge

An H-bridge formation allows us three different load voltage levels:

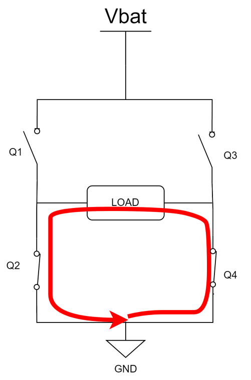
1) +Vbat - when Q1 and Q4 are open.
2) -Vbat - when Q2 and Q3 are open.
3) 0 - when Q2 and Q4, or Q1 and Q3 are open.



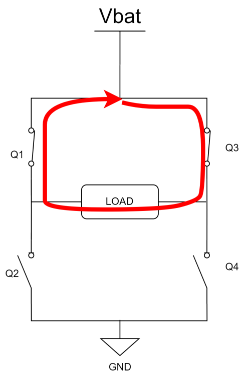
H-bridge - load voltage is +Vbat



H-bridge - load voltage is -Vbat



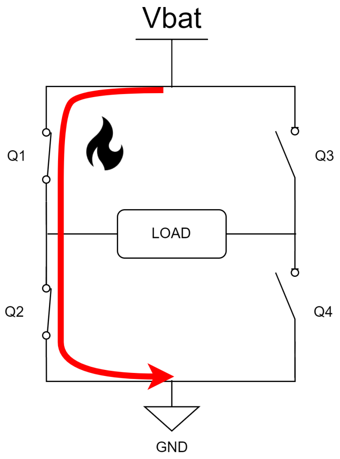
H-bridge - load voltage is 0



H-bridge - load voltage is 0

When alternating between each diagonal pair of switches, the polarity of the load voltage changes respectively, resulting in a perfect square wave of $\pm\ Vbat$.

In a typical H-bridge application, these switches are implemented using N-MOSFET transistors, each allowing current flow at the same direction (from battery to ground) when open. The transistors operate like a digital switch - open whenever their gate is supplied with high voltage, and close otherwise, not allowing current to go through. Usually, the timings of switch opens are orchestrated by either integrated circuits ("MOSFET drivers"), an MCU, or both. The reason for that is that H-bridge is prone to damaging the circuit if not handled carefully, as any constellation in which both switches of the same side are open, causes a short circuit.



H-bridge - short circuit

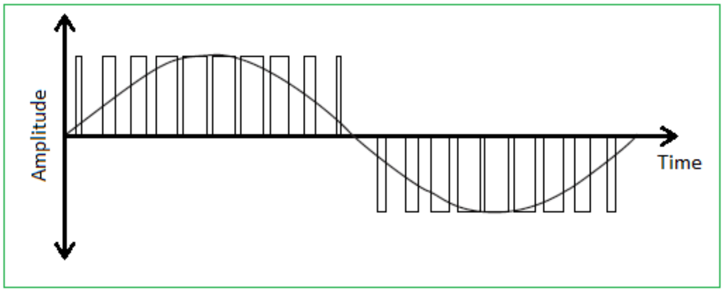
Not only is the energy conversion efficiency of an input square wave low, it can also damage certain electronic devices, which expect a pure sine AC wave as their power input. For these reasons, this naive square wave modulation is not used by power supply systems such as the UPS. The H-bridge formation, though, can be used with a smarter modulated input, called **PWM** (pulse width modulation).

AC generation

Using PWM, the H-bridge can be controlled with pulses in varying lengths, which allows the formation of a much more analog-looking, pure sine wave. This type of circuit is called a PWM inverter, and the theory behind it is out of this paper's scope, its basic foundations can be reduced to two relevant outcomes:

1) It uses pulse signals with varying width (i.e. change of voltage from low to high, back and forth after a varying delay).
2) Controlling the pulse width is equivalent to controlling the instantaneous output signal. It offers a programmable API to "draw" the output signal by controlling the PWM sequence.

To modulate a sine wave, the PWM signal in a PWM inverter uses two polarity levels, representing the two halves of a sine wave, and the equivalent two halves of an H-bridge which control the direction of the current. It is then chained to a transformer to generate a pure AC signal:



PWM of a sine wave. This is used as the input voltage of the transformer in an inverter H-bridge.

The transformer integrates the PWM pulses to construct an analog sine wave, while also amplifying the sine amplitude from the battery DC level (~25 Volt) to the required AC level of the power grid.

Inverter load feedback

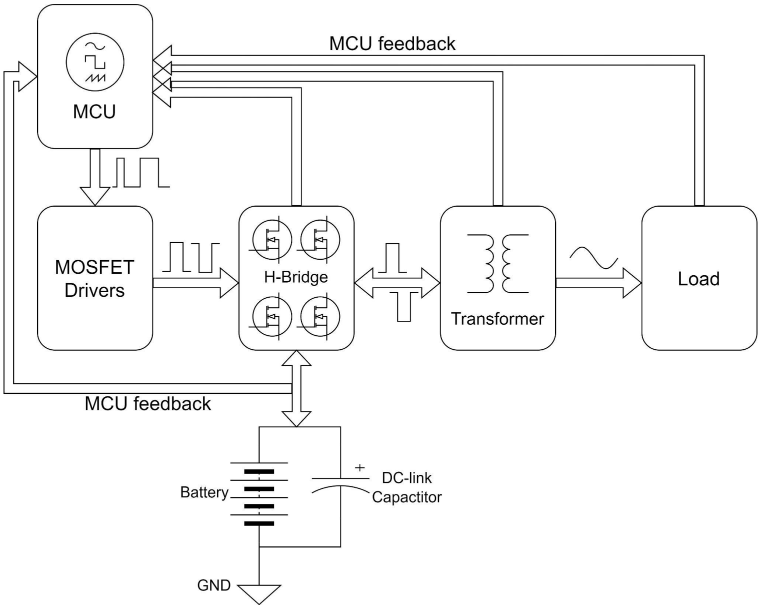
One of the challenges in real-life inverters is that the load voltage supplied by the inverter ultimately fluctuates due to possible changes in the connected load - a very massive load can lead to a decrease in output voltage level or the opposite way. A PWM inverter can be an effective tool to counter the effect of varying load. In modern UPS implementations the MCU which controls the PWM signal, will also constantly measure the actual output voltage level in a feedback-loop. This feedback is used to fine-tune the PWM signal, to constantly counteract changes on the inverter's load, as well as to implement various safety measures.

DC link capacitor

One of the disadvantages of the PWM inverter design is that the pulses are formed using rapid sharp changes between voltage levels. These sharp changes trigger current spikes that can damage the circuit. In order to protect the circuit and "absorb" these current spikes, a very large capacitor is connected between the DC source and the ground.

Full model

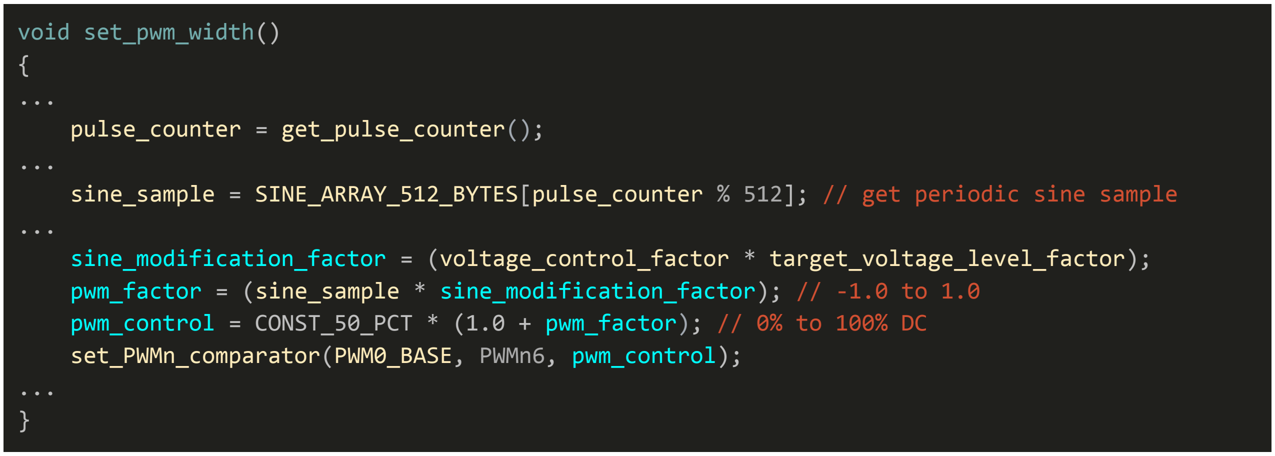
These building blocks, in similar variations, are formed together to create the basic model of a modern DC to AC inverter. A detailed diagram of power regulation blackbox introduced at the beginning of this section can be seen below:



Modern PWM inverter schematic design

Altering the power output

Having learned the basics of how a UPS implements the DC-to-AC conversion mechanism, we reverse-engineered the relevant code paths within the TI MCU, that is responsible for controlling the PWM signal, while reacting to the feedback signal of the transformer's output voltage. The code below implements the feedback loop that ensures fine tuning of the PWM signal, by comparing measured output signal to a desired constant sine wave, and then correcting the PWM signal by the difference between the two. The fine-tune control of the PWM signal is made by multiplying a dynamic calculated factor representing the feedback loop (labeled *voltage_control_factor* in the code below), with a constant perfect sine wave, representing the expected output signal. The sine wave samples are stored as a constant 512-byte array, consisting of values of a digitized period of a sine wave from 1.0 to -1.0. The offset (phase) within that array progresses cyclically at each PWM cycle.



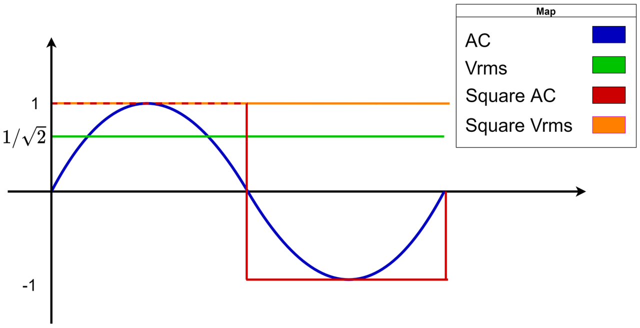
```
void set_pwm_width()
{
...
    pulse_counter = get_pulse_counter();
...
    sine_sample = SINE_ARRAY_512_BYTES[pulse_counter % 512]; // get periodic sine sample
...
    sine_modification_factor = (voltage_control_factor * target_voltage_level_factor);
    pwm_factor = (sine_sample * sine_modification_factor); // -1.0 to 1.0
    pwm_control = CONST_50_PCT * (1.0 + pwm_factor); // 0% to 100% DC
    set_PWMn_comparator(PWM0_BASE, PWMn6, pwm_control);
...
}
```

set_pwm_width function. Note that SINE_ARRAY_512_BYTES controls the general periodic shape of the signal.

Some devastating impacts can be achieved by changing that array only.

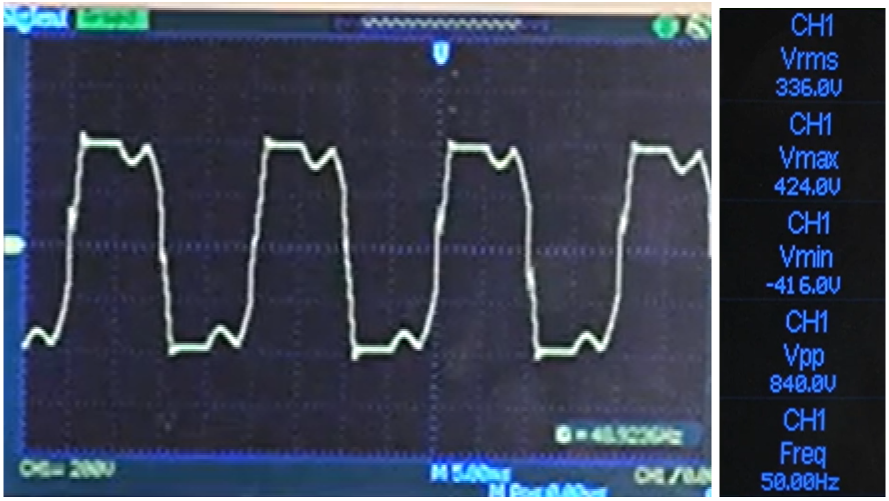
Over Voltage

To damage the UPS downstream devices, two possible parameters of an AC signal can be altered - the voltage, and the AC frequency. Our first attempt was to attempt altering the output voltage, trying to output the highest voltage the UPS may supply. The trivial effort is taking the transformer to its limits. We patched the voltage destination value (used as part of the inverter load feedback logic) to its maximum value and reached the limit of the transformer - around 300 Volts. In order to get the voltage to an even higher level, we abused the fact that an AC signal with a pure sine wave will deliver a lower voltage than an AC signal with a square wave that has a similar amplitude. This is due to the fact that an AC voltage is actually the "absolute average" voltage of the AC signal (it is calculated with Vrms - which stands for **root mean square voltage**), and so a square wave with the same amplitude as a sine wave, will have a higher "average" voltage, as demonstrated below:



Overlaid square wave and sine wave with RMS calculations

In order to achieve that, we patched the constant signal shape in the MCU firmware to a square wave of the same period. The result is just as expected - a Vrms of 336Volts, beyond the transformer "AC" limits.

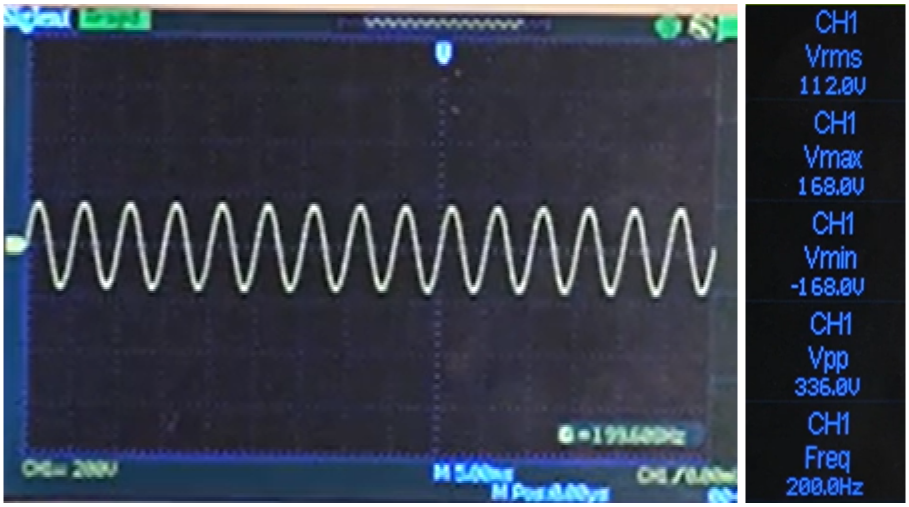


UPS output of square wave taken from oscilloscope. Vrms reached 336 Volts.

Higher Frequency

Another manipulation of the output signal may be increasing the output frequency. Higher voltage frequencies have a similar potential to damage the UPS downstream devices.

By simply changing the sine shape from including a single period to two, we get an output signal of a doubled frequency. The same can be done to multiply the frequency by 4, 8, or any other power of two.



UPS output of high-frequency signal taken from an oscilloscope. The frequency is 200Hz instead of 50Hz.

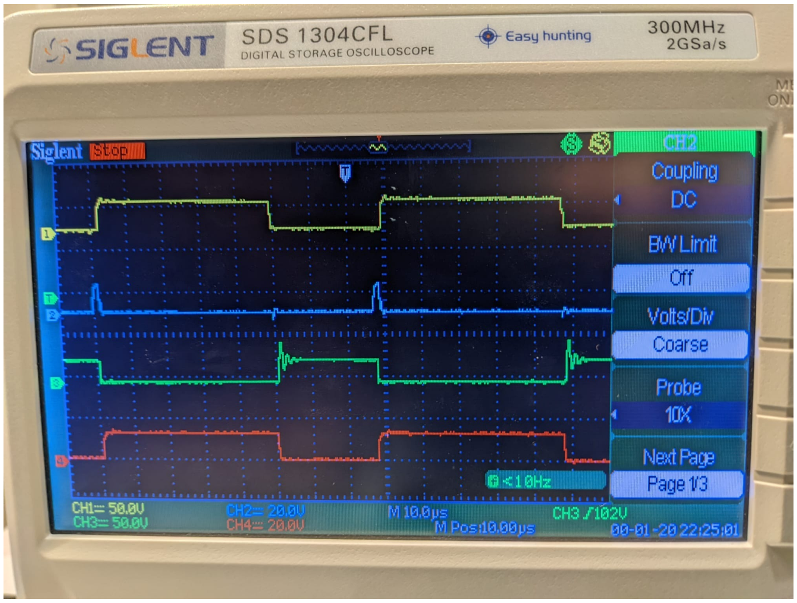
DC link capacitor burst

Another approach to abuse the UPS is focusing on the UPS itself, rather than its downstream devices. By controlling the electrical circuits, some edge cases can be reached that may physically damage certain electrical components within the UPS. The immediate candidate for such damage is the DC link capacitor.

The DC link capacitor is the weakest link in a PWM inverter, almost by design. As explained in the section above, the circuit is designed in a way which makes the DC capacitor "absorb" extreme currents in the circuit, and these currents cannot be avoided in a PWM inverter. The suitable DC capacitor to place in the circuit is meticulously chosen to support the extreme currents, given a known set of edge cases the circuit is expected to produce (as detailed in this article, for example).

Though calculated carefully, the edge cases are still assumed to be within the constraints of a functioning inverter, accompanied with some backup software checks which are not handled by the analog circuit. By controlling the power regulation MCU, extreme edge cases can be achieved while the subsequent software mitigations are disabled.

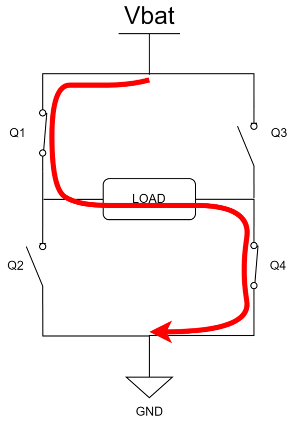
The idea is to use the MCU code for malicious control of the H-bridge gates. The H-bridge is connected to an integrated circuit which supplies hardware protection to the bridge from short circuits (i.e, ensuring no high voltage is supplied to both FETs of one bridge side at the same time). However, it does not protect it from other undefined cases. One of these cases occurs when only one of the four gates is open, keeping the circuit open and forcing zero current on the transformer.



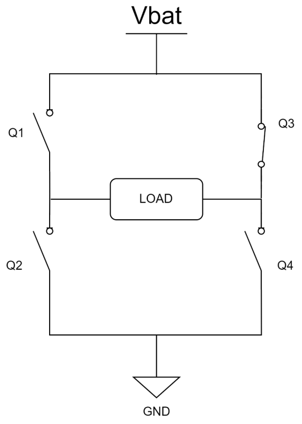
Gate voltage on FETs Q1-Q4

Using our software control of the TI MCU, we set a periodic transfer between two states:

1) Gates 1 and 4 are on, the voltage on the transformer is forced to be Vbat.
2) Only Gate 3 is on, the transformer is in an open circuit state, with zero current.

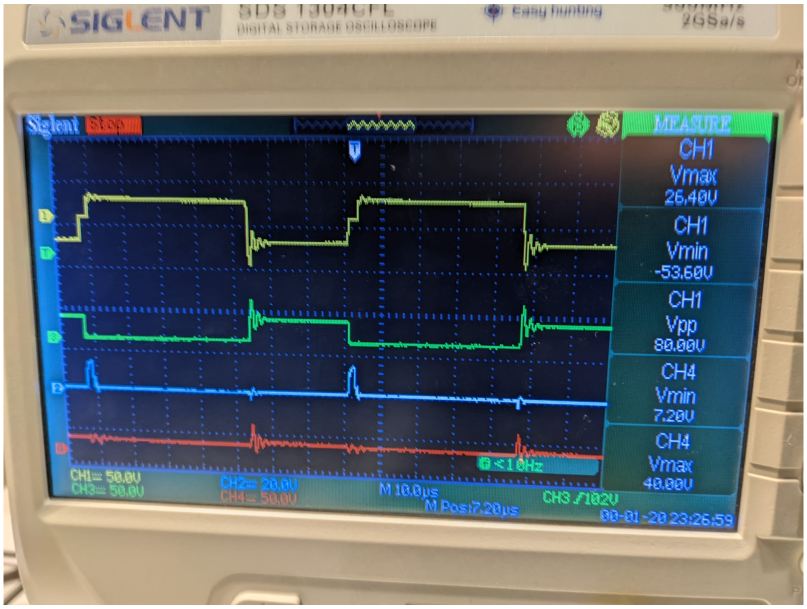


State 1 - Vbat load voltage



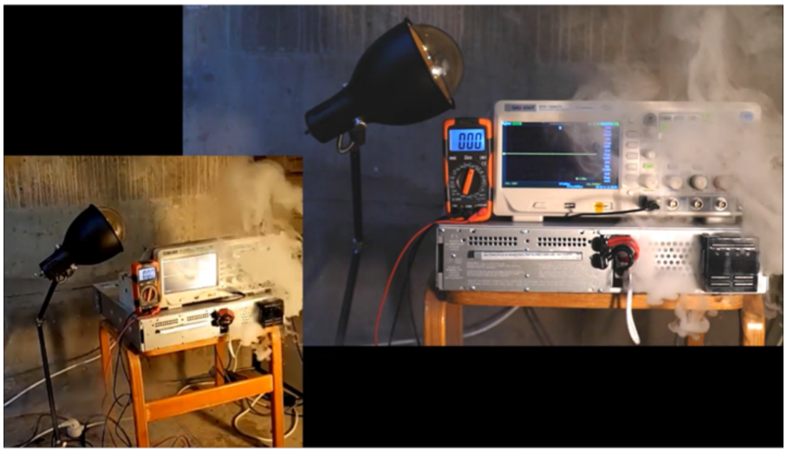
State 2 - open circuit

For transformers, the voltage is correlative to the change in current. The immediate cut-off of current results in a very low transformer voltage:



Transformer voltage, Q3, Q2, battery voltage, from top to bottom

The results are dramatic. As can be seen from the oscilloscope snapshot the transformer voltage decreases to a negative peak of -53.6V, and the total change in voltage is almost 80V in approximately 1us! This dramatic change propagates to other parts of the circuit, as well as a temporary fluctuation on the H-bridge supply battery, between 7V to 40V (the nominal voltage of the battery is 25V). The DC link capacitor is connected in parallel to the battery, and the rapid changes in voltage cause enormous current spikes. For a capacitor of 2700uF, as the one used by the Smart-UPS, a voltage change of 33V in 1usec interval develops a current spike of almost **100KA!** The current spike was so high that the MCU triggered a short-circuit alert and turned off the UPS. However, by using our RCE vulnerability we were able to bypass the software protection and let the current spike periods run over and over until the DC link capacitor heated up to ~150 degrees celsius (~300F), which caused the capacitor to burst and brick the UPS in a cloud of electrolyte gas, causing collateral damage to the device.



UPS in smoke after overheating the DC link capacitor

Final notes

The discovered vulnerabilities emphasize the risk that arises from the transition to “Smart” cloud connected devices. Advances in connectivity features should go side by side with adoption of modern security standards to ensure the safety of the end users. Safety, as this research demonstrates, is a literal concern when dealing with cyber-physical systems. The exploitation risk is no longer limited to the IT world - an attacker can turn the UPS to a physical weapon. From a cyber security point of view, these kinds of systems must be handled as a flammable substance that sits in the heart of an organization.

The security strategy for connected devices of that nature should involve independent layers of protections. Physical analogue (not digital) protections must be implemented with a software agnostic solution as much as possible, accompanied with the common software mitigations perfected in the IT world. As customers, we would not approve an unsigned Windows or Mac upgrade while at the same time, many connected devices vendors don’t have any signing mechanism for their firmwares and the firmware upgrade process is easily accessible with minimum authentication. We believe that the reason for that is lack of public awareness and hope that the risk shown in this paper will push vendors to deploy firmware signing mechanisms and make the “unsigned firmware vulnerability” a thing of the past.

Another important observation derived from this research applies to implementation of secure transportation protocols. We demonstrated how a faulty implementation of a glue logic “pseudo layer” can overshadow the proper implementation of either underlying or overlying logical layers, even if implemented carefully. In this case, it was shown how an innocent misconception of responsibilities diffusion between the APC implementation and Mocana’s NanoSSL library TLS implementation, left a connection hanging, prone to state confusion vulnerabilities which can be escalated to remote-code-execution. These APIs must be inspected with extra caution and tested thoroughly, as the entire communication is always as vulnerable as its most vulnerable layer.

While the discovered vulnerabilities are now patched or mitigated, there are probably some other vulnerabilities in similar devices. As smart devices tend to communicate with their management servers over the Internet even if the cloud features are rarely used, the Internet becomes a part of the device attack surface. This underlines the need for identification and anomaly detection of network activity for connected physical systems.