ARMIS

TLStorm 2.0

A set of critical vulnerabilities for Aruba and Avaya switches that can break network segmentation

Gal Levy
Yuval Sarel
Noam Afuta
Barak Hadad

Table of Contents

Introduction

In March 2022, we disclosed TLStorm1.0 – a set of critical vulnerabilities in APC Smart-UPS devices. The vulnerabilities allow an attacker to take control over Smart-UPS devices from the internet with no user interaction and make the UPS literally go up in smoke. The root cause for these vulnerabilities was a misuse of NanoSSL, a popular TLS library by Mocana.

Using the Armis knowledgebase—a database of over 2 billion devices and over 6 million device profiles—we were able to identify dozens of devices using Mocana NanoSSL, including two popular switch vendors that are affected by the same misuse of the NanoSSL library.

These vendors are Aruba (acquired by HP) and Avaya Networking (acquired by Extreme Networks). We have found that both vendors have switches vulnerable to remote code execution (RCE) vulnerabilities that can be exploited over the network.

This research details our new findings, dubbed TLStorm 2.0, which include vulnerabilities that could allow an attacker to take full control over these switches. The exploitation of these RCE vulnerabilities has some severe implications discussed in this paper, such as breaking of network segmentation, allowing lateral movement, data exfiltration to the Internet and captive portal escape.

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:

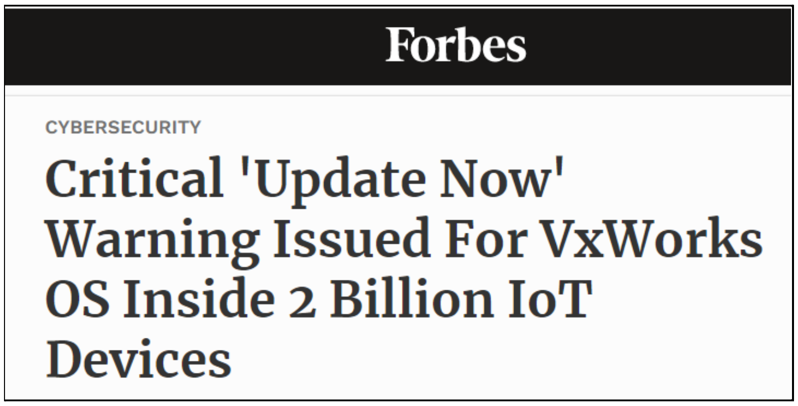
- TLStorm - Three critical vulnerabilities discovered in APC Smart-UPS devices which can allow attackers to remotely manipulate the power of millions of enterprise devices.
- PwnedPiper - Nine vulnerabilities in Swisslog Pneumatic Tubes System (PTS) - a critical infrastructure used in hospitals
- Modipwn - 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).
- BlueBorne - An attack vector targeting devices via RCE vulnerabilities in Bluetooth stacks used by over 5.3 Billion devices.

External Libraries as an entry point

While the use of external libraries comes with a lot of advantages, like in-house development time reduction, or using a third party code written by developers coming from a specific field of expertise, it also brings with it a built-in uncertainty. Implanting a “foreign” code, means the user is fully trusting it to be implemented safely, since any security issues originated in the external library, are embedded within it, and automatically become security issues which the user has less control over.

These external libraries are shared between various product lines, making them a target for attackers - finding one breach in one library can “reward” the attacker with a series of attacking tools for many targets.

We could get a sense of the potential scale of finding such a software supply chain in previous URGENT/11 research, where the vulnerable TCP/IP stack (IPNet) was part of the supply chain of many IoT operating systems like VxWorks, INTEGRITY and more, seeding a monstrous tree of inherited vulnerabilities, putting billions of endpoints at risk eventually.



Forbes

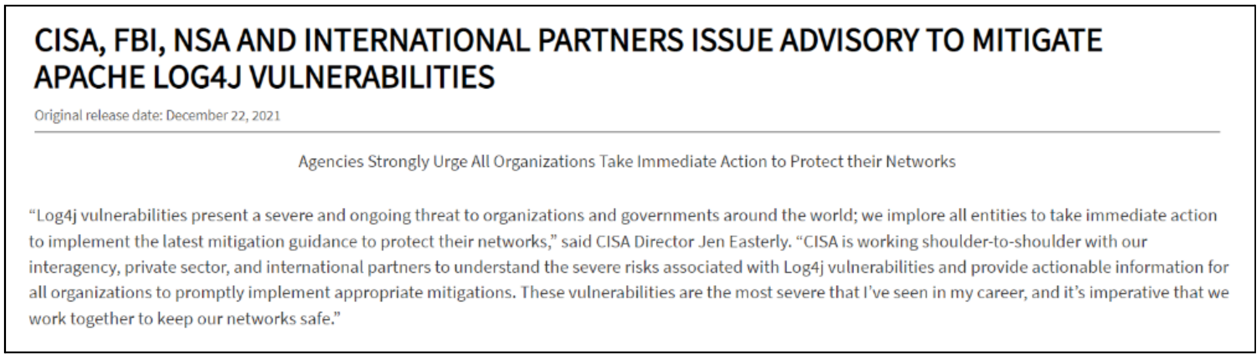
CYBERSECURITY

Critical 'Update Now' Warning Issued For VxWorks OS Inside 2 Billion IoT Devices

Forbes article covering URGENT/11, reporting of 2 Billion vulnerable IoT devices

The Framwork4shell vulnerabilities

Probably the two most significant security disclosures of the previous year were log4shell and spring4shell, both originated in an external library used widely by Java code frameworks. These two were a textbook use case for the risk of a bug in an external library, putting millions of users at risk for using a seemingly innocent external library. In these cases, both libraries were open source, and the critical bugs were lying inside the code for years unnoticed. These events were a painful reminder for the community of the significance of flaws in widely used libraries.



CISA, FBI, NSA AND INTERNATIONAL PARTNERS ISSUE ADVISORY TO MITIGATE APACHE LOG4J VULNERABILITIES

Original release date: December 22, 2021

Agencies Strongly Urge All Organizations Take Immediate Action to Protect their Networks

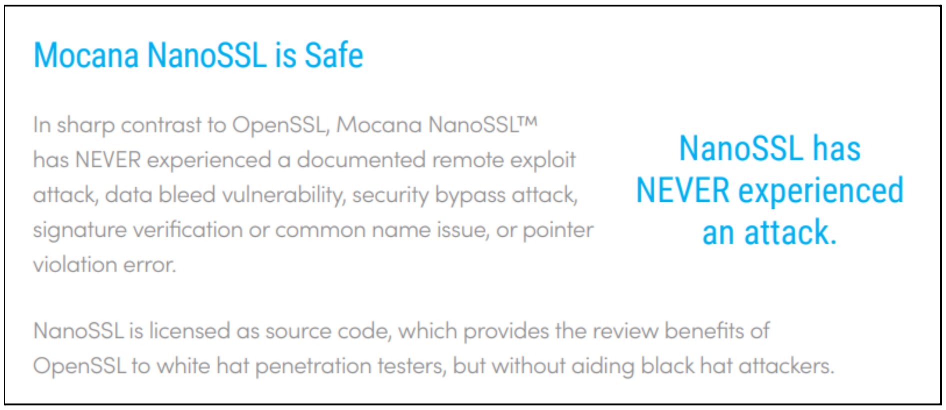
“Log4j vulnerabilities present a severe and ongoing threat to organizations and governments around the world; we implore all entities to take immediate action to implement the latest mitigation guidance to protect their networks,” said CISA Director Jen Easterly. “CISA is working shoulder-to-shoulder with our interagency, private sector, and international partners to understand the severe risks associated with Log4j vulnerabilities and provide actionable information for all organizations to promptly implement appropriate mitigations. These vulnerabilities are the most severe that I've seen in my career, and it's imperative that we work together to keep our networks safe.”

Official warning by the CyberSecurity & Infrastructure Security Agency of the USA

NanoSSL

In the TLStorm research we demonstrated how an attacker was able to exploit the NanoSSL library created by Mocana, a company that supplies third party cryptographic solutions, acquired lately by DigiCert.

Mocana highlights its advantage as a close source library, comparing to the main actor of ssl libraries, the famous OpenSSL library,



Mocana NanoSSL is Safe

In sharp contrast to OpenSSL, Mocana NanoSSL™ has NEVER experienced a documented remote exploit attack, data bleed vulnerability, security bypass attack, signature verification or common name issue, or pointer violation error.

NanoSSL has NEVER experienced an attack.

NanoSSL is licensed as source code, which provides the review benefits of OpenSSL to white hat penetration testers, but without aiding black hat attackers.

Taken from a white paper written by Mocana

However, advanced research methods do not require access to source code. Moreover, the library itself does not even have to include critical bugs to experience attacks by bad actors. The risk from external sources is not limited to bugs in its implementation, a misuse of the library may put it in an unintended state that the library was not meant to handle. Especially when close source libraries are used, and the user is “blind” to the implementation itself, the user must implement a glue-logic code that connects its code to the external library. If the user does not follow the instructions of the external API tightly, critical bugs can be exploited even if both ends of the glue-logic are flawlessly implemented. We have demonstrated an example of that in the TLStorm1.0 whitepaper.

Inspired by the disastrous impact caused by the log4shell vulnerability, we understood the potential risk of the NanoSSL bug as part of a widely used external library. Though the root cause of the NanoSSL bug comes from a misuse of the API rather than a straight forward bug in the package, it still indicates a developers’ approach that might recur in other vendors, in the very same exploitable manner. In other words, any vendor using NanoSSL relying on similar wrong assumptions as the ones taken by APC, is vulnerable to TLStorm attacks.

```
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, ignoring the return value
            coap_client_ops->mocana_ssl_recv_message(
              coap_client,
              pbuf);
          tcp_recved(tpcb, pbuf->tot_len);
          pbuf_free(pbuf);
...
}
```

Reminder - APC misuse of NanoSSL. *mocana_ssl_recv_message* return value is ignored.

NanoSSL is widely spread among vendors, and known for being one of the most trust-worthy third party TLS solutions. Besides Schneider Electric, other industry giants are also identified as Mocana's clients, among them are Siemens, IBM, HP, Lenovo and more. Armis knowledge-base allows us a simple heuristic for pairing visible TLS stack to a device with a set of other identifiers, and by that we set out to find various NanoSSL usages. In this research we focused on two instances of NanoSSL found in popular network switches. There are surely many more we did not explore.

Vulnerable Switches

Before we dive into the vulnerabilities themselves we should first cover the security incentives for researching network switches, both by attackers and defenders. These incentives are based on the concept of "network segmentation".

Network Segmentation

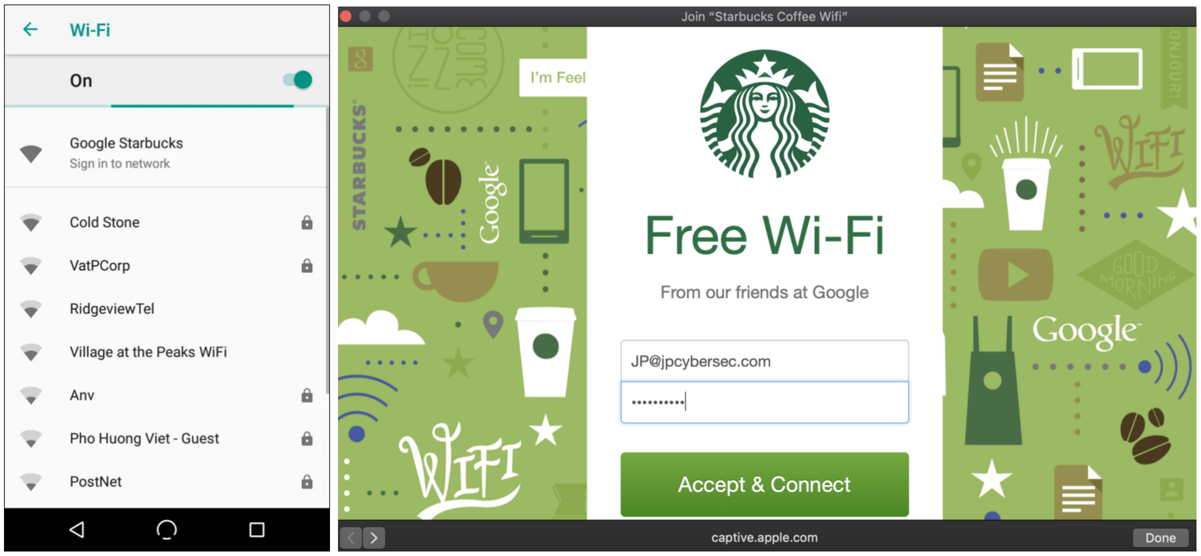
Every network is composed by the devices (endpoints) that use it, but also by the devices that create its backbone - network appliances like routers and switches. These devices are often overlooked when examining the security concerns of an organization even though they themselves implement the concept of sub-network isolation (i.e. network segments, or VLANs). This virtual concept of splitting the network into segments is the first line of defense against attackers trying to perform lateral movement.

To shed some light on this concern, let's explore a specific way of enforcing network segmentation, which later in this paper will be a feature of interest.

Captive Portal

A captive portal is a web page displayed to new users after connecting to a network (usually using Wi-Fi), acting as a portal to the network. There, admins most commonly prompt users with some requirements to enter the network, like authentication, payment or agreement to a policy or terms.

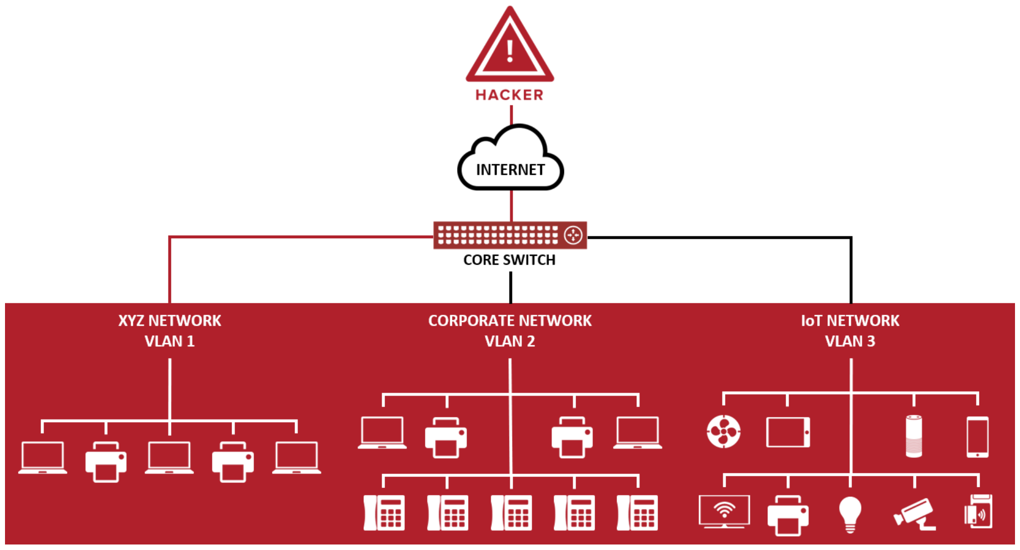
Such portals are most common in networks created for “casual” guest users, such as airport lounges or shopping malls. Such users are held “captive” in this tiny segment of the network (comprising only the user and the access point) until they can achieve the needed criteria for entrance to the network. The captive portal is protecting the organizations needs, whether they are financial, legal or security focused.



Example of captive portal at a starbucks store, taken from #1 and #2

CDPwn

Another example exploring the crucial role network equipment and network segmentation have in keeping the network protected is another research by Armis Labs - CDPwn. In that research we found several vulnerabilities in layer 2 protocols in popular network equipment, and explored the implication they have. For further reading please refer to the CDPwn whitepaper.



Example from CDPwn, flattened network segmentation

Combining what we've learned from CDPwn about network segmentation and its weak spots, and the vulnerabilities we've found by tracking the NanoSSL libraries supply chain traces, we've found several critical vulnerabilities in popular network equipment and leveraged them to some severe implications.

Discovered vulnerabilities

Recap - APC SmartConnect Vulnerabilities

As mentioned, on March 8th, we disclosed 3 critical vulnerabilities in APC's UPS devices. 2 of the vulnerabilities originated by a misuse of Mocana's TLS library NanoSSL:

CVE-2022-22806

TLS authentication bypass: A state confusion in the TLS handshake leads to authentication bypass, leading to remote code execution (RCE) using a network firmware upgrade.

CVE-2022-22805

TLS buffer overflow: A memory corruption bug in packet reassembly (RCE). This vulnerability can be triggered whenever a user of NanoSSL does not close the TLS connection after SSL_recv returns an error code, and proceeds to call SSL_recv again afterwards. The simple nature of this vulnerability and the high likelihood of a developer making this easy-to-miss mistake when using an external library's API, inspired us to search for more devices that use Mocana's NanoSSL, and to see if they were indeed also vulnerable to the same attack.
The vulnerabilities in this research paper, along with the vulnerabilities in the previous TLStorm research paper, highlight how a flaw in a single library means every device using said library could be potentially vulnerable to the same attack concept.

Avaya/ExtremeNetworks

This newly found set of vulnerabilities is exposed via the same attack surface - the switch's web management portal. Accessing the web portal's landing page does not require authentication in itself, meaning it allows any user in the network to establish TLS communication, and set up a pre-authenticated secured peer. Such pre-authenticated communication leads to various flows of HTTP POST packets processing, where we found three critical vulnerabilities. The nature of this direct pre-authenticated network access elevates the severity of these vulnerabilities to the most critical level, allowing a zero-click attack with a very low complexity.

But first we need to understand the attack surface better, and specifically the bottleneck of every single incoming byte from the TLS communication peers - the routine ***mocana_recv_wrapper***.

```
int mocana_recv_wrapper(void * ssl_sock, char * p_out_buff, int buff_size)
{
...
    errno = mocana_tls_recv(..., p_out_buff, buff_size, &bytes_received, ...);
    if ( errno >= 0 )
      return bytes_received;
    return errno;
...
}
```

mocana_recv_wrapper, return value represents negative errno, or bytes received if positive.

Worth noting that this routine returns a signed integer value - negative for error, positive for successful, representing the amount of bytes read. If the read is finished, the return value of zero is expected, representing a successful “zero bytes read”. Some of the vulnerabilities occur because of mishandling of an erroneous return value.

CVE-2022-29860 (9.8 CVSS score) – TLS reassembly heap overflow

This vulnerability is another instance of TLStorm vulnerability, and more precisely it is a variation of CVE-2022-22805. It perfectly highlights how the very same vulnerable concepts we explored in the APC research apply to other NanoSSL users.

```
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 >= 0x4800 )
         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;
}
```

Reminder - SSL_receiveRecord routine, includes TLS packets reassembly functionality

If this code looks familiar, it is just because it was already presented in CVE-2022-22805 explanation. It is the very same code used in the APC, with a single difference which is the maximum size of ***expected_len*** of 14832 (0x4800) bytes instead of 2389. This is the same NanoSSL library compiled with different configuration. One way or another, **the vulnerability concept is the same - if there is a piece of code that ignores the return value, we will be able to exploit the reassembly heap overflow, the exact same way we have already proven to be feasible in the previous research.**

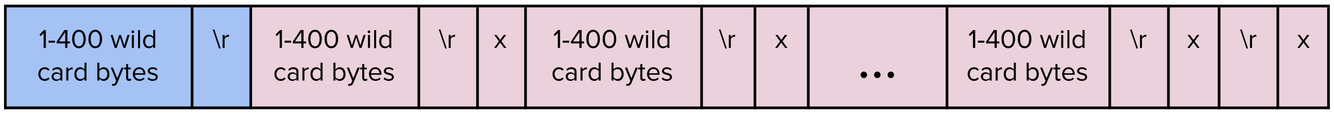
The most simple scan of this function usage shows that our concerns were proven right. The following flow inside ***handle_POST*** process ignores the return value of ***mocana_recv_wrapper***:

```
int parse_POST_content_header(void *tls_connection, ...)
{
START:
  received_buf[0] = 0;
  bytes_recveived = 0;
  while ( 1 )
  {
    // get bytes until reach size limit or ‘\r’
    result = mocana_recv_wrapper(tls_connection, &received_buf[bytes_recveived], 1);
    if ( result <= 0 )
      return result;
    ++bytes_recveived;
    ...
    if ( bytes_recveived == 400) // if reached max size 400
        return -105; // return matching error
    if ( received_buf[bytes_recveived - 1] == '\r' )
    {
      // return value is not checked
      // can launch TLStorm and set malformed exceeding size
      mocana_recv_wrapper(tls_connection, waste_byte, 1); // consume following ‘/n’
      ...
      if ( !strlen(received_buf) ) // supposed to be met when ending with “\r\n\r\n”
        return 0;  // return success
      goto START;  // receive next HTTP header component
    }
  }
}
```

parse_POST_content_header, reads header components in loops, ignores mocana return value in between

This function consumes HTTP header components in a loop, each component at a time for up-to 400 bytes, or until **‘\r’** is reached. After reaching **‘\r’**, we can see that the code attempts to carelessly consume another byte expecting for **‘\n’** (**‘\r\n’** line suffix), validating neither the content nor the return value. An attacker can send a huge trailing packet instead of the expected **‘\n’** byte. This launches the TLStorm attack, setting the next TLS packet expected size to a huge size without reallocating the heap buffer.

To actually overwrite the heap, we will need to keep consuming bytes into the non-reallocated buffer. We do that by shaping the sent data in **‘\r’** separated chunks of up-to 400 bytes. We can stop the override by ending our buffer with “**\r x \r x”** 4-byte sequence, where x is a wild card byte (**‘\n’** is not validated). Overall the malicious TLS stream layout looks as follows:



Malicious TLS stream layout

In blue, is the first TLS packet of the stream, followed by a pinkish TLS packet of size larger than the maximum NanoSSL hardcoded size of 0x4800 bytes. We can see that the attacker has close to complete flexibility in the choice of override bytes (still needs to insert **'\r'** between every 1 to 400 bytes), making this primitive ideal for RCE exploitation.

CVE-2022-29861 (9.8 CVSS score) – HTTP header parsing stack overflow

This vulnerability exists due to an improper boundary check - a buffer is allocated statically to a fixed length of 200 bytes, and later used as a source buffer for memcpy to another 200-byte stack buffer, but the size of the copy operation can be manipulated to be 204, or even more since the copied string is not guaranteed to be null terminated.

The vulnerable flow is triggered when processing HTTP headers when submitted with enctype **"multipart\form-data"**:

```
void handle_POST_multipart_form_data(...)
{
...
  get_http_boundary(received_http_buf, &in_str);
  if ( !receive_tls_and_cmp_str(ssl_sock, in_str, ...) )
  {
     ...
  }
}
```

handle_POST_multipart_form_data

First, the boundary value is extracted from the received HTTP header inside *get_http_boundary_function*:

```
char * get_http_boundary(char *received_http_buf, char **in_str)
{
...
    boundary = zalloc(200);  // allocate and zeroise
    if ( boundary)
    {
      // Find location of “boundary=” field
      if ( strncmp("boundary=", html_header, 9) )
      {
         ... // set html_header value accordingly
      }
      ...
      // find start of boundary received value
      start_of_boundary = find_char(html_header, '=') + 1;
      ...
      do
      {
        ++size;  // increase size until first appearance of ‘\r’, ‘\0’, or ‘;’
      }
      while ( start_of_boundary[size] != '\r' ||
              start_of_boundary[size] != 0 ||
              start_of_boundary[size] != ';' );
      ...
      if ( size <= 200 ) // size is exactly 200 if ‘\r’, ‘\0’, or ‘;’ at the 201st
byte
      {
          memcpy(boundary, start_of_boundary, size);
      }
...
    return boundary;
}
```

handle_POST_multipart_form_data

The function is called with a parameter that contains the HTML headers received by the server. This parameter points to a string allocated on the heap, which is attacker controlled from the network. The string is allocated and populated in a function which is responsible for fetching the ***boundary*** parameter from the header received from the web. As can be seen in the decompilation of the function, the boundary value string has a size limit of 200, and if the stop condition is met exactly in the 201st byte, the buffer is fully occupied with the received boundary value, and if the matching byte is **‘\r’** or **‘;’** rather than zero, the occupied string will not be null-terminated. This will turn out to be determinantal soon.

The following function to be called after extracting the boundary is *receive_tls_and_cmp_str*:

```
int receive_tls_and_cmp_str(void * ssl_sock, unsigned __int8 *in_str, ...)
{
  char buf[200]; // [sp+8h] [-E8h]

  str_len = strlen(boundary) + 4; // set received tls copy length
  ...
  received_bytes = 1;
  ...
  do
  {
      // receive TLS into the stack buffer
      bytes_recveived_1 = mocana_recv_wrapper(ssl_sock,
                                              &buf[received_bytes],
                                              str_len - received_bytes);

      retval = bytes_recveived_1;
      if ( bytes_recveived_1 < 0 )
        return retval;
      received_bytes += bytes_recveived_1;
  }
  while ( received_bytes < str_len );
...
}
```

receive_tls_and_cmp_str

We can see here that the function starts with calculating the TLS reception length, and for this calculation it adds 4 to the length of the freshly extracted boundary string. However, we already know that an attacker can set this string to not be null-terminated, which means that the length value will depend on the first appearance of **'\0'** in the heap after the boundary buffer. Having this null terminator in a distance of at least 40 bytes from the end of the boundary value, allows an attacker to override the stack deep enough to reach the Link Register (LR) with TLS controlled data, and gain RCE. This could be achieved by heap shaping, or probably by statistically spraying the vulnerable payload enough times.

The exploitation requires first sending HTTP POST with boundary value of the full 200-byte size:

"boundary="	boundary value - 200 bytes not including '\r', '\0', or ';'	'\r' or ';'

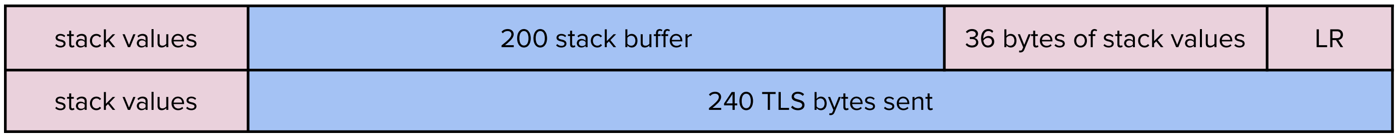
Malicious boundary value

This value should be placed before heap buffer not containing zeroes for at least 40 bytes, to form a valid string of size larger than 240 bytes:

some heap buffer	200-byte boundary value	1..40 or more non zero bytes	'\0'

Heap layout in exploit flow, boundary value string length is larger than 240 bytes

And this state should be followed by another TLS packet of at least 240 bytes:



Stack layout in *receive_tls_and_cmp_str*, before and after call to *mocana_recv_wrapper*

By overwriting LR, gaining a remote code execution is a relatively simple task, using return-oriented-programming (ROP) technique or another info leak vulnerability.

Vuln 3 - HTTP POST request handling heap overflow

This vulnerability also originates from a mishandling of Mocana return value. In this case, the return value is not fully ignored, but simply assumed to be successful only. The ***handle_POST*** routine calls ***mocana_recv_wrapper*** in loop with a stop condition of zero return. I.e, finished reading, and every non-zero return value is treated as a valid read bytes amount, including negative error values:

```
void handle_POST(ssl_socket_t * ssl_socket, avaya_struct_t * avaya_struct){
...
      // calculate the malloc size, per user controlled data
      malloc_size_errno = malloc_size_per_type(..., &malloc_size, ATOI_TYPE);
      if ( malloc_size_errno >= 0 )
      {
        // Allocate the buffer per input allocation size plus one
        buff = malloc(malloc_size + 1);
        if ( buff )
        {
          if ( malloc_size )
          {
            tot_len = 0;
            while ( tot_len < malloc_size )  // success condition
            {
              next_len = mocana_recv_wrapper(ssl_socket,
                                             &buff[tot_len],
                                             malloc_size - tot_len);
              if ( !next_len )  // stop condition, when zero is received
                goto FINISH;
              tot_len += next_len;  // next_len can be a negative mocana errno
            }
...
```

handle_POST, bytes are consumed and accumulated until *mocana_recv_wrapper* returns zero

We can see here the ***mocana_recv_wrapper*** return value handling bug, and ***tot_len*** is accumulated until a zero return value is reached (end of data). To exploit this bug, we need to understand where ***buff*** - the destination buffer of ***mocana_recv_wrapper*** - originates from. We can see that ***buff*** is dynamically allocated during this function, with an allocation size of ***malloc_size + 1***. ***malloc_size*** is user controlled, in the full 32bit range from 0 to 0xFFFFFFFF (signed int “-1”). When inserting input ***malloc_size*** “-1”, the actual allocation is eventually using input size 0 (0xFFFFFFFF + 1). This device's particular malloc implementation fixes allocation size 0 to 1, and succeeds:

```
int malloc_most_inner(..., int size, ...)
{
  if ( size == 0 )
    size = 1;
...
}
```

malloc_most_inner, malloc size zero is fixed to one, practically operates as malloc(1).

After passing the allocation validation successfully, the TLS reception loop is being performed. In each iteration an attempt to read up to "malloc_size - tot_len" bytes of tls packets takes place. The first iteration will attempt to read up to 0xFFFFFFFF bytes. However, there's a boundary check inside Mocana tls receive function which validates that the signed recv size is positive:

```
int mocana_tls_recv(..., int max_recv_size, ...)
{
...
  if ( max_recv_size <= 0 )
    return -6010;
...
}
```

mocana_tls_recv, returning -6010 errno if max_recv_size is negative

This function will fail, returning -6010. However, as seen in the ***handle_POST*** function, the returned error is ignored and treated as the number of bytes read.

Than ***tot_len*** is being updated:

$$tot\_len = tot\_len + next\_len = 0 - 6010 = -6010$$

The next iteration will attempt to read up to "malloc_size - tot_len" again. This time the calculation is:

$$read\ size = -1 - 6010 = -6011$$

The read attempt will fail again with a boundary check of mocana_tls_recv, returning -6010 again. The read size will now be:

$$read\ size = -6011 - 6010 = -12021$$

After N iterations, the read size will be:

$$read\ size\ =\ \ -1\ -\ 6010N$$

The read size calculation will recur until its sign flips back to positive after an underflow, which happens exactly after $FLOORED(INT\_MIN\ /\ 6010)\ =\ 357319$ iterations.

After the underflow occurs, the signed value of the read size will be a huge positive number (just below MAX_INT 0x7FFFFFFF), resulting in a valid tls buffer size read. Though it seems to be a “too large” size to read, the actual read size is fully controlled, since the huge calculated value represents only the upper limit of the TLS reception. Then the TLS content overrides the 1-byte allocated heap buffer, with attacker controlled data of attacker controlled length which can lead to remote code execution (RCE).

End-of-life vulnerability

This vulnerability, unlike the others, is found in a device family which has reached end-of-life, meaning a patch for this vulnerability is not, and will not be supplied. These devices, though, can still be found in the wild, per data acquired from Armis knowledge base. The risk of end-of-life devices is too easy to disregard, while it poses the very same risk as supported unpatched devices, which usually attracts the main focus of security offices. We will discuss this issue later in the document.

Aruba/HPE

CVE-2022-23677 (9.0 CVSS score) – NanoSSL misuse on multiple interfaces (RCE)

This vulnerability is another Mocana NanoSSL related vulnerability - caused by the same vulnerable concepts shown in the TLStorm research for the APC Smart UPS, and CVE-2022-29860 on the ExtremeNetworks switches shown in this research paper.

Like the others, this vulnerability is caused by the TLS reassembly state confusion first shown in CVE-2022-22805 on the APC UPS, and the mishandling of error codes by the code using the NanoSSL API. This vulnerability, along with the others in this paper, highlight how a flaw in a single library means every device using said library could be potentially vulnerable to the same attack concept.

Let’s take a closer look at this vulnerability. The vulnerable flow occurs when the switch attempts to parse an incoming HTTPS request from a user on a port with Captive Portal redirection enabled. The purpose of this parsing code is to create a redirect HTTP response to the URL of the Captive Portal server. The following is the vulnerable function. After the TLS handshake is concluded, it is called in a loop, with each iteration triggered when new data is written to the TCP buffer.

```
http_request_content * parse_incoming_http(...)
{
    ...
    if ( is_session_ssl )
    {
        // Receive a single TLS record using mocana
        rc = SSL_recv_wrapper_mocana(...);
            // if Mocana recv fails, close the connection
            if ( rc < 0 )
            {
              ...
              return mocana_closeConnection_wrapper(...);
            ...
            }
        // While there is still data in the TCP buffer - continue calling mocana SSL_recv
        do
        {
            ...
            // When there is no more data in TCP buffer, rc is error code (< 0)
            rc = SSL_recv_wrapper_mocana(...);
            ...
        }
        while ( !rc );
            // Loop stops once error code is received, but connection is not closed in case of error.
      }
        ...
        // check for '\r\n\r\n' to stop receiving new data and start parsing http packet
        ...
  return result;
}
```

Simplified decompilation of function that receives HTTPS data and parses it to create redirect URL

As we can see in the code above, there are 2 calls to Mocana's *SSL_recv* function. Each call to the function reads a single TLS record, and decrypts it. The first call's return value is checked, and in case of error the connection is closed. The second call is within a loop. It is called as long as there is data to be read in the TCP buffer. Once the TCP buffer is emptied, the *SSL_recv* function immediately returns with error code *ERR_TCP_READ_ERROR* and the loop exits. However - the exact erroneous value of the return value is not checked, and regardless of the type of error, the connection is not closed. This is very similar to the other Mocana TLS reassembly vulnerabilities we've found. This means that if an attacker can trigger the vulnerable flow in the second call to *SSL_recv*, they can set up the vulnerable state confusion, and once the calling function *parse_incoming_http* is called again for additional data, the first call to *SSL_recv* overwrite the Mocana heap buffer, similarly to the other vulnerabilities we've shown. Let's elaborate on how this can be achieved.

As we've mentioned before, each call to *SSL_recv* reads a single TLS record. If there is no data to be read in the TCP buffer, it exits immediately with *ERR_TCP_READ_ERROR*. Therefore, in order to trigger the state confusion in the second call to *SSL_recv,* we must make sure there is data in the TCP buffer at that point.

To achieve that, an attacker could send 2 TLS records in a single TCP packet. Consider the following TCP buffers:

TCP packet A:

TLS Record		
Type	Length	Data
23 (Application Data)	len(Data)	...

Initial TLS record

TLS Record	
Type	**Length**
23 (Application Data)	0x5000 (any size bigger than 0x4800)

Malicious TLS record

TCP packet B:

0x5000 bytes of attacker controlled bytes
PWNPWNPWN...

The first TLS record in *packet A* will be received and decrypted successfully in the first call, and since there is more data in the TCP buffer, the second call will be triggered to parse the second, malicious TLS record of *packet A*. The second call will reach the vulnerable code in the inner function *SSL_recvRecord*. If an attacker intentionally fails the size check inside *SSL_recvRecord (using size > 0x4800)* they can trigger the state confusion and cause *SSL_recv* to return with an error. The details of the state confusion itself are in the previous TLStorm whitepaper - CVE-2022-22805.

Since the erroneous return value is not checked, the connection will not be closed. The next valid packet sent will trigger the first call to *SSL_recv*, and allow an attacker to overwrite the Mocana heap buffer with as many user-controlled bytes as written in the malicious second record's TLS header.

CVE-2022-23676 (9.1 CVSS score) – RADIUS client memory corruption vulnerabilities

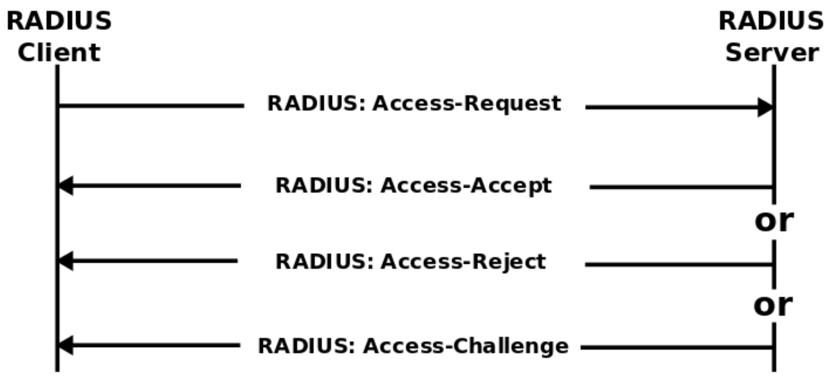
This vulnerability is in the RADIUS client implementation of the switch. First, a few words about RADIUS

RADIUS

RADIUS is an authentication/authorization protocol that provides centralized authentication. RADIUS is a client/server protocol, with the client sending requests to authenticate users or machines, and the server checking the credentials and info of the user and responding with acceptance or rejection of the request.

A typical access attempt transaction involving RADIUS will usually go like this:

1. A user attempts to gain access to a network server with a set of credentials
2. The network server *sends* an *Access Request* message containing information about the user requesting access, and the credentials it used to the RADIUS server
3. The RADIUS server verifies the credentials of the user and responds with on of the following:
    a. Access Accept
    b. Access Reject
    c. Access Challenge - requesting more information. The RADIUS client responds to this message with another Access Request



Radius protocol - from [wikipedia]

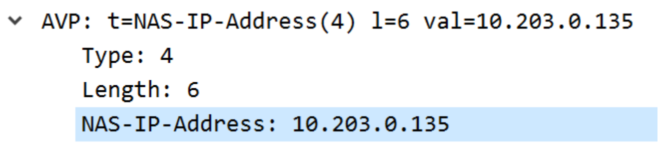
The RADIUS server and client hold an agreed upon shared key used for encrypting password and sensitive information in the packet, and signing the messages with the Message Authenticator field.

RADIUS packets can also carry within them EAP (Extensible Authentication Protocol) data, indicating the EAP authentication should be performed.

RADIUS Client Vulnerabilities

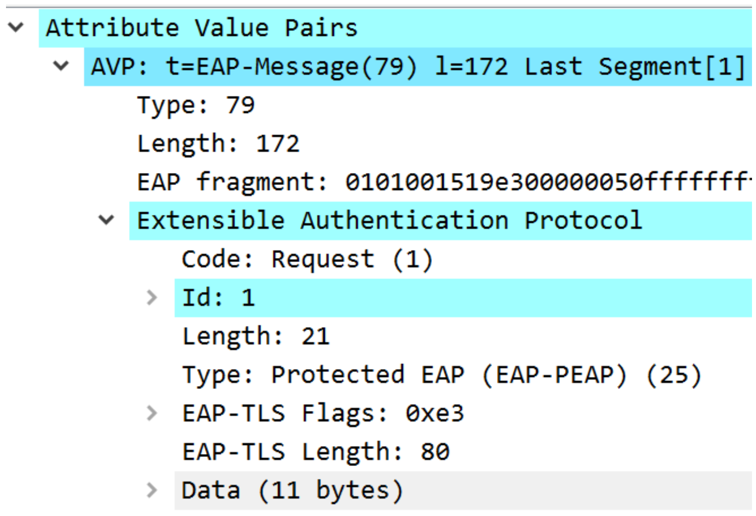
The vulnerable flow occurs when an access attempt is made to the switch, the switch sends an Access Request packet to the RADIUS server, and the server responds with an Access Challenge RADIUS packet. The attack surface is accessible when handling any type of access attempts to the switch. This includes, SSH, telnet, web, etc.

The vulnerability itself is in the PEAP (an encrypted version of EAP) parsing code of the switch's RADIUS client. A RADIUS packet's data is a list of AVPs - Attribute Value Pairs. Each of these contains the AVP's type, its length and the data itself. For example, this is the AVP containing a user's IP address:



```
⌄ AVP: t=NAS-IP-Address(4) l=6 val=10.203.0.135
    Type: 4
    Length: 6
    NAS-IP-Address: 10.203.0.135
```

When using EAP authentication, an EAP-Message AVP is added to the message. The data of this AVP is the content of the EAP fragment itself. EAP is a protocol in and of itself and the EAP fragment's header contains a length as well. This is an example of the RADIUS AVP and the EAP fragment encapsulated inside:



```
v Attribute Value Pairs
  v AVP: t=EAP-Message(79) l=172 Last Segment[1]
      Type: 79
      Length: 172
      EAP fragment: 0101001519e300000050ffffff
    v Extensible Authentication Protocol
        Code: Request (1)
      > Id: 1
        Length: 21
        Type: Protected EAP (EAP-PEAP) (25)
      > EAP-TLS Flags: 0xe3
        EAP-TLS Length: 80
      > Data (11 bytes)
```

The vulnerability is caused by mishandling of the two different size fields.

Let's take a look at the vulnerable code:

```
int get_eap_message_from_packet(radius_packet_t *radius_packet)
{
  ...
  // extract the EAP_Message AVP from the RADIUS packet
  eap_attribute = (EAP_message *)get_attribute_from_packet(
                                    &radius_packet->attributes,
                                    EAP_Message,
                                    ...);
 ...
    // extract the length from the EAP fragment header - the “inner” length
    eap_message_inner_length = eap_attribute->eap_fragment.length
    if ( eap_message_inner_length - 4 < 0xBFD )
    {
      // allocate size according to the EAP header length (inner length)
      eap_message_buffer = malloc(eap_message_inner_length);
      ...
      // copy bytes from packet - use the size from the AVP (outer length)
      memcpy_(eap_message_buffer, &eap_attribute->eap_reassembled, eap_attribute->length - 2);
      ...
}
```

get_eap_message_from_packet - simplified decompilation of function that extracts EAP content from RADIUS packet

When attempting to extract the EAP fragment from inside the RADIUS AVP, the code above allocates a buffer for the data with size from the EAP header. It then uses memcpy to copy bytes into this buffer, using the size from the AVP header - 2. It does this because normally the RADIUS header length is always EAP header length plus 2, since it also contains in it the AVP type and length fields. However, since the switch never verifies that these sizes are actually the expected values, or rather - the expected size difference, an attacker can send a small size in the EAP header, and a large size in the RADIUS header, causing the overflow of the allocated buffer.

For example - the following packet buffer will cause a heap overflow of the *eap_message_buffer*

RADIUS headers	AVP header (EAP Message)		EAP fragment				
...	Type	Length	Code / ID	Length	Type	...	Data
...	79 (EAP-Message)	255	...	5	26 (PEAP)	Attacker controlled headers	Attacker controlled data

Malicious RADIUS packet

When attempting to parse the example packet above, the switch will allocate a buffer the size of 5 bytes, and then proceed to copy 255 bytes from the EAP fragment into the buffer. These bytes are almost entirely attacker controlled and will overwrite the heap - eventually allowing for remote code execution.

Since RADIUS is an authenticated protocol, and the packets are “signed” using a pre-shared secret between the client and server - an attacker would either need access to the RADIUS server, or to the shared secret.

Exploiting the NanoSSL bug - Examples

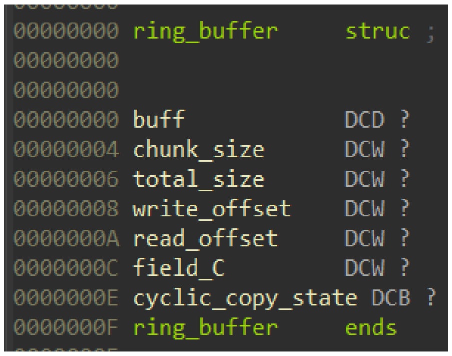
APC CVE-2022-22805 exploitation

Exploiting this vulnerability quickly came across a challenge in the form of the usage of the heap across the codebase. Using a combination of static analysis and debugging we noticed that the heap is barely used after initialization. This means that in order to exploit this vulnerability “classic” heap overflow exploitation approaches like overriding the heap chunks headers and leveraging the heap management to perform write-what-where won’t work. Combining this with the fact that **after** the overrun buffer there isn’t any “interesting” data structure or pointer, we had to find a different approach.

Since the RAM doesn’t implement any Read-Write-Execute restriction, we decided to look at what exists after the heap. There we were able to find a data structure that enabled a primitive which was escalated to full RCE exploitation.

Inter-Processor-Communication

The OS implementation is based on recurring mainloop in addition to a bunch of asynchronous interrupts that update global values used by the mainloop, and so on. However, the code infrastructure still uses some common building blocks, for a safe implementation of more convoluted objectives, like communication with external hardware and peripherals. The main structure for representing the memory buffers delivered between different interfaces is ring buffers. This structure is crucial for understanding practically everything - from the bootloader firmware update, to the physical voltage control (which will come in handy later on). The exploitation uses those ring buffers as well.



```
00000000 ring_buffer     struc ;
00000000
00000000
00000000 buff            DCD ?
00000004 chunk_size      DCW ?
00000006 total_size      DCW ?
00000008 write_offset    DCW ?
0000000A read_offset     DCW ?
0000000C field_C         DCW ?
0000000E cyclic_copy_state DCB ?
0000000F ring_buffer     ends
```

Ring buffer structure

In every OS code iteration, the state of the different hardware interfaces is checked by reading those buffers, and changed by filling them. If the ring buffer state indicates memory corruption, the code clears their memory and initializes the ring buffer state.

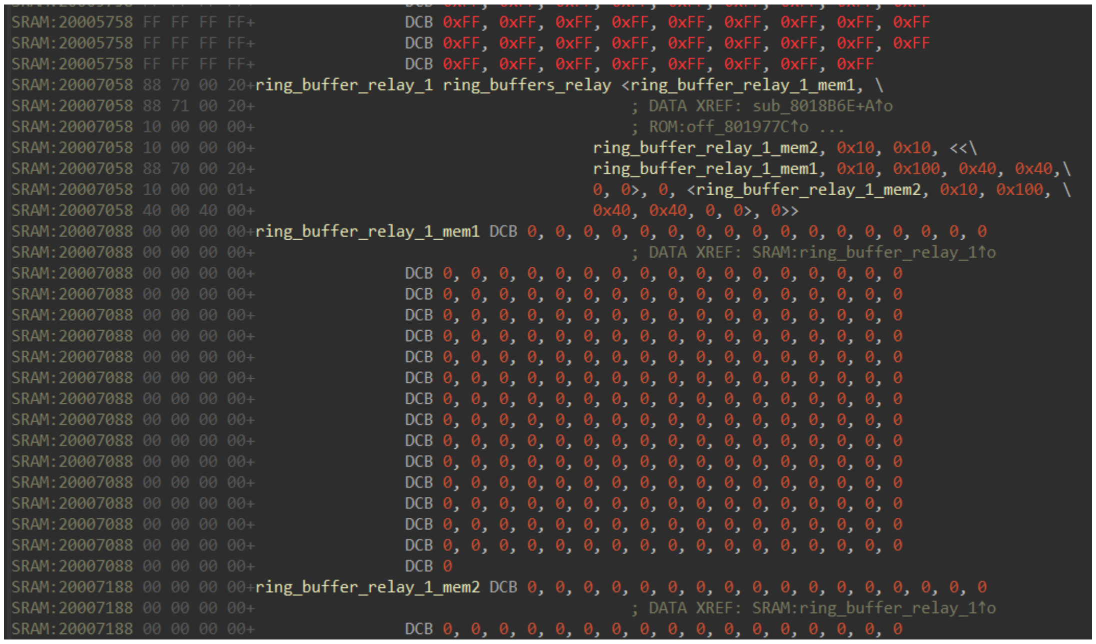
```
void read_ring_buffer_chunk(ring_buffer *ring_buffer, char *out_buff)
{
  if ( ring_buffer->cyclic_copy_state <= 2 ) // Validate ring buffer state
  {
     ... // cyclic copy of buffer
  }
  else
  {
    init_ring_buffer_state(ring_buffer); // init the ring buffer if state is invalid
  }
}
```

read_ring_buffer_chunk

```
void __fastcall init_ring_buffer(ring_buffer *ring_buffer)
{
  total_size = ring_buffer->total_size;
  progressing_index = 0;
  ring_buffer->write_offset = 0;
  ring_buffer->read_offset = 0;
  ring_buffer->cyclic_copy_state = 0;
  while ( progressing_index < total_size )
    ring_buffer->buff[progressing_index++] = 0;
}
```

init_ring_buffer

As we can see, a cyclic_copy_state larger than 2, is invalid and triggers ring buffer re-initialization. These ring buffers representations are found right after the end of the heap.


```
SRAM:20005758 FF FF FF FF+                DCB 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
SRAM:20005758 FF FF FF FF+                DCB 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
SRAM:20005758 FF FF FF FF+                DCB 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
SRAM:20007058 88 70 00 20+ring_buffer_relay_1 ring_buffers_relay <ring_buffer_relay_1_mem1, \
SRAM:20007058 88 71 00 20+                                     ; DATA XREF: sub_8018B6E+A↑o
SRAM:20007058 10 00 00 00+                                     ; ROM:off_801977C↑o ...
SRAM:20007058 10 00 00 00+                                   ring_buffer_relay_1_mem2, 0x10, 0x10, <<\
SRAM:20007058 88 70 00 20+                                   ring_buffer_relay_1_mem1, 0x10, 0x100, 0x40, 0x40,\
SRAM:20007058 10 00 00 01+                                   0, 0>, 0, <ring_buffer_relay_1_mem2, 0x10, 0x100, \
SRAM:20007058 40 00 40 00+                                   0x40, 0x40, 0, 0>, 0>>
SRAM:20007088 00 00 00 00+ring_buffer_relay_1_mem1 DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                                     ; DATA XREF: SRAM:ring_buffer_relay_1↑o
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007088 00 00 00 00+                DCB 0
SRAM:20007188 00 00 00 00+ring_buffer_relay_1_mem2 DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
SRAM:20007188 00 00 00 00+                                     ; DATA XREF: SRAM:ring_buffer_relay_1↑o
SRAM:20007188 00 00 00 00+                DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

ring buffers memory layout

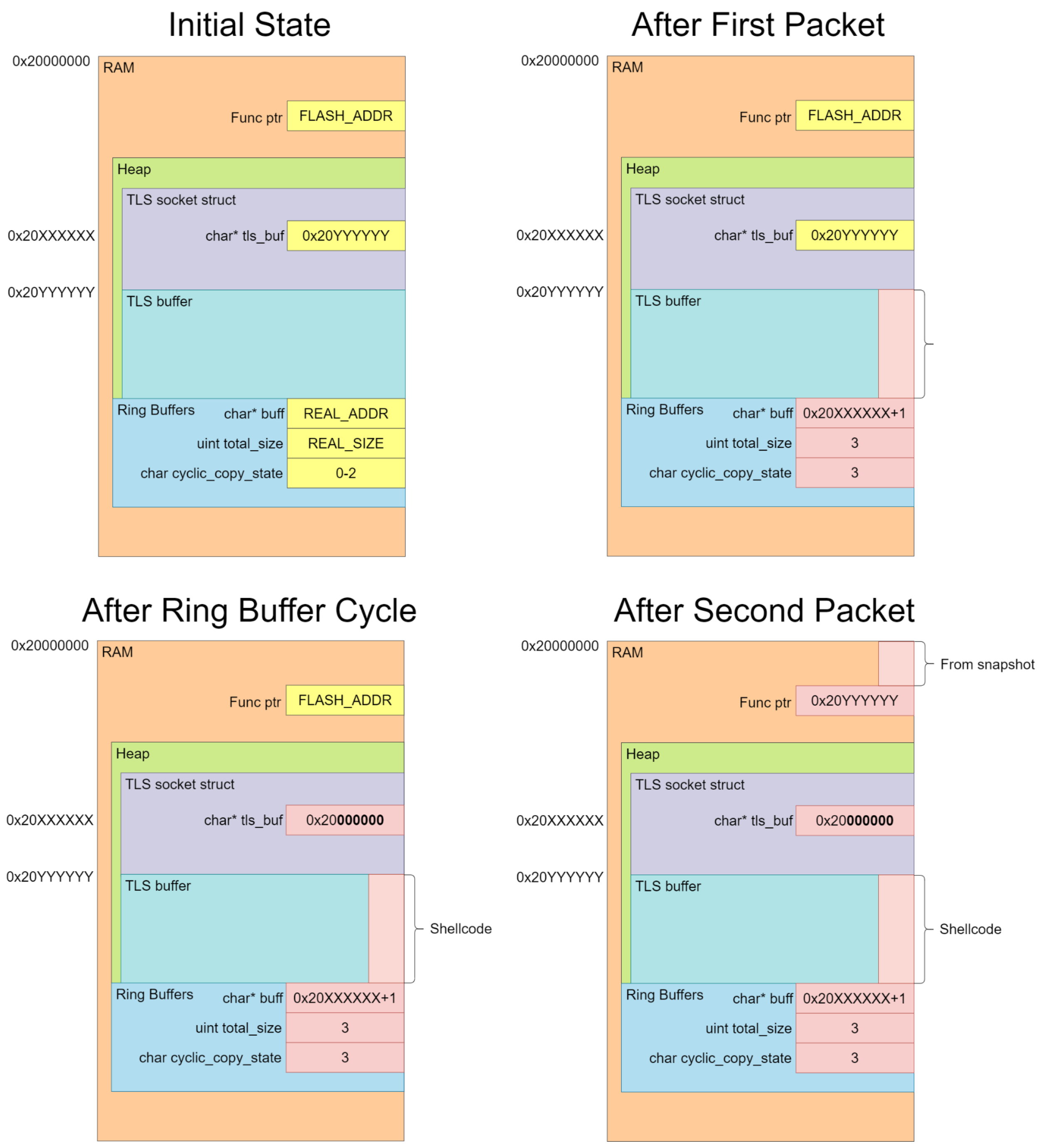
Our strategy was to override the first ring buffers relay structure, and just wait for the mainloop to start a chain reaction. The ideal goal would be to find a write-what-where primitive using the ring buffer content relay. We couldn't find a way to achieve that, since the relay is directed to another memory representation defined in another structure (which we won't discuss here) which is not attacker controlled - we could either copy to our controlled ring buffer from another constant memory source address, or the opposite direction. In other words this means that if we want to use the relay ring buffer functionality, we either control the copy destination but not its content, or the copy content but not its destination. This can't be escalated to a useful primitive.

Alternatively, we chose to use the ring buffers initialization functionality. As shown earlier, when the ring buffer state is undefined, the ring buffer is reinitialized - a routine which initializes the header and writes zeros to the ring buffer's memory buffers. When overwriting the ring buffer state to an undefined value, the ring buffer memory pointer to X, and the ring buffer memory size to N, we gain a "write N-zeros to X" primitive - which means that a controlled amount of sequential zeros can be written to wherever we want in the RAM. This is a strong primitive, and the path to fully exploiting this vulnerability is at hand.

Utilizing "write-N-zeros-where"

The network buffer which is at the center of this vulnerability is controlled by us, and is not being reallocated. Our plan is to utilize our new primitive to override with the 3 LSBs of the destination buffer pointer (which is located at a fixed address). After that override has taken place, sending a new buffer from the same connection will override another RAM memory rather than the heap. More specifically, the buffer pointer will point to the start of the RAM at address 0x20000000.

The content of the new buffer will contain a dump of a pre-fetched running device's memory as a template, and change only the content of the first function pointer in RAM, which will be the end of our overriding buffer. The first function pointer in that particular case happens to be a function pointer that is being called spontaneously by the mainloop. Since the address of the execution is now attacker controlled, it can be set to point to the huge unused buffer of the overwritten heap, as the memory space for our shellcode (which we should have sent in the first packet).



Memory map after every exploitation step

To summarize:

1) Trigger the heap overflow - a huge buffer that contains a shellcode and overrides the entire unused heap and the following first ring buffers relay structure.
2) Wait for the mainloop to use the patched ring buffers relay structure, and clear the LSBs of the ssl sent buffer destination pointer.
3) Send another huge buffer that overrides the start of the RAM - containing a memory dump of the same device's RAM start, followed by a patched function pointer that points to the beginning of the sent buffer from step 1.
4) Wait for the mainloop to call our patched function pointer.

Aruba CVE-2022-23677 exploitation

Heap overflow turned remote code execution

This vulnerability also leads to remote code execution. Considering this is a heap overflow, we started out with 2 possible directions in mind:

- Overwriting data in the heap that might lead to a stronger primitive. Looking for function pointers, addresses to user data buffers or any other lead. While there were some strong candidates, we didn't find a way to deterministically overwrite them without corrupting other critical data. We found that, in this case, this approach would be feasible with an additional info leak primitive.
- Overwriting the heap meta-data in order to elevate our writing primitive. This proved to be the better direction to take.

Heap format

The heap is made out of memory blocks. When a call to malloc takes place, a block of an appropriate size is chosen and allocated. To keep track of the heap's blocks, their sizes, and which are free or allocated, the switch's OS uses a linked list of free heap blocks. Each free block starts with a header of 3 fields: Block size, pointer to next block, and pointer to previous block.

Size	Next block	Previous block	Empty data space (size - header_size bytes)

Header of each free heap block

To find a block to allocate, the malloc code traverses this list until it finds a big enough block to satisfy the user's request. If the block is bigger than the requested size, it splits into 2 pieces - the first remains free, and the second is allocated and returned to the user.

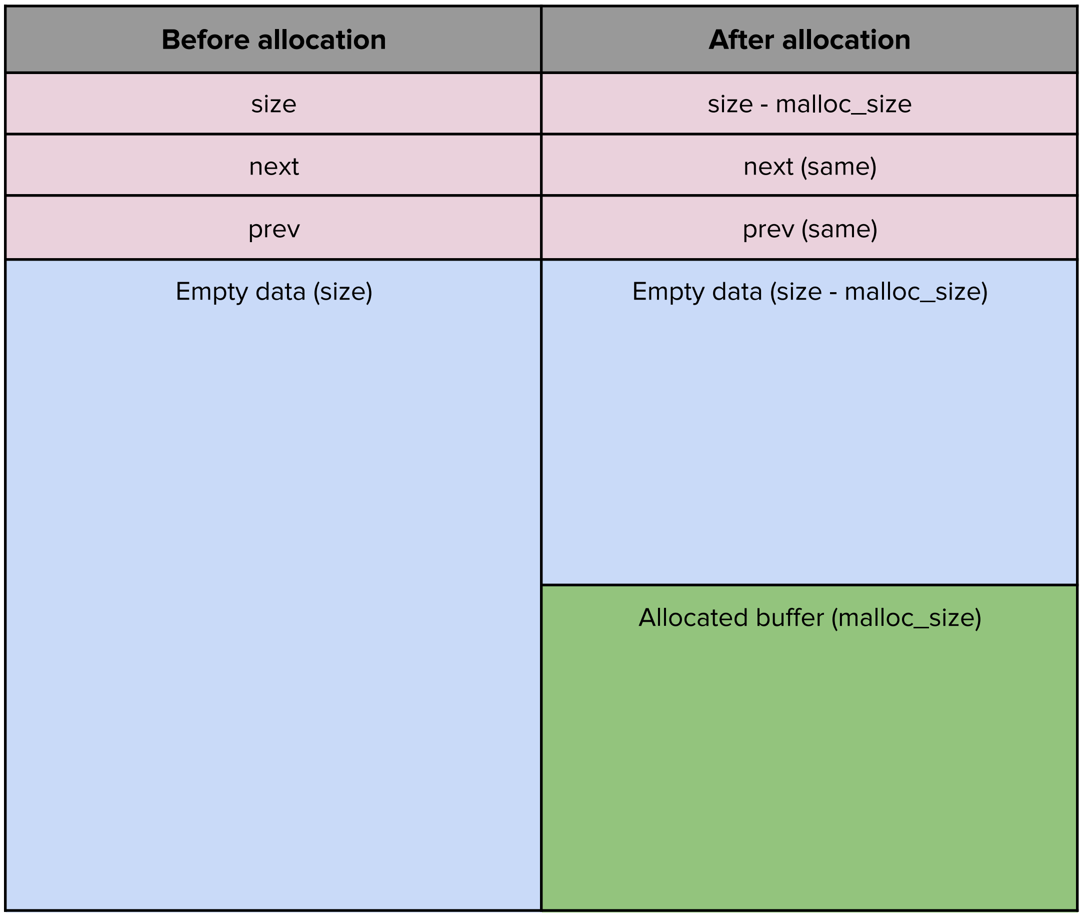


Illustration of a free heap buffer before and after allocation

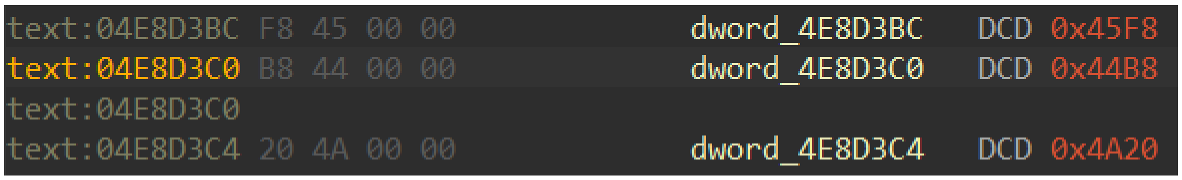
We now understand that newly allocated buffers are almost always followed by a free buffer header.

Malloc the text section

Since we're exceeding the boundaries of our heap buffer, we can also overwrite the next free heap block following our allocated block. This gave us the interesting prospect of overwriting the *size* and *next block* fields of the block header.

Based on this, the exploitation plan goes as follows - overwrite the *size* field with a value of our choosing, and overwrite the *next block* field with an address in the code area of the memory where our malicious code will eventually be executed. Then send a packet to the switch that will trigger a heap allocation. Since our buffer will be bigger than the size we wrote to the overwritten heap header, the malloc code will check the next free buffer, which will now point to the code area we chose. If the *"size field"* of that header is big enough, the area will be successfully "allocated", and any data in the packet will be written directly to the code area of the memory. The size check means that we have to choose an address in memory that contains a "*size*" big enough for the buffer.

To find candidates for good overwrite addresses, we wrote a script to find suitable sizes in the .text section (code) of the memory.



```
text:04E8D3BC F8 45 00 00                dword_4E8D3BC   DCD 0x45F8
text:04E8D3C0 B8 44 00 00                dword_4E8D3C0   DCD 0x44B8
text:04E8D3C0
text:04E8D3C4 20 4A 00 00                dword_4E8D3C4   DCD 0x4A20
```

Example of address we used in our “fake” heap header

Consider the following example. We overwrite the heap header following our block and write the address 0x4E8D3C0 (as seen in the image above) to the *next block* field. We also write a size smaller than 0x45f8 to the *size* field. Then, when we send a packet with the size 0x45f8 to the switch, the “free block” in address 0x4E8D3C0 will be allocated, and the data in our packet will overwrite the code following this address. For example, we found a candidate that allowed us to overwrite the code of *malloc*. Once any task in the switch calls *malloc*, the shellcode will be triggered. This is practically game over. We can fix any corruption of the heap linked list, execute a shellcode, and restore overwritten code or data.

This technique proved to be efficient, succeeding at a reasonable rate. However, it is still a statistical exploitation, because of possible scheduling issues such as context switches between the allocation of the “code section buffer” and the memory overwrite. Still, we deemed it good enough to show that this threat is not theoretical..

Final notes

Both phase one (APC Smart UPS vulnerabilities) and phase two (Network equipment vulnerabilities) of the TLStorm project emphasize the risk of vulnerabilities deeply embedded in the supply chain of common software libraries which end up in different devices across different industries. For NanoSSL, a common misuse made completely different codebases to be vulnerable to almost identical vulnerabilities.

NanoSSL was supposed to act as a security feature, acting as a TLS layer and inhibiting malicious actors from accessing sensitive communications. But instead, the misusages detailed in this research made NanoSSL the entry point from which an attacker was able to take control over the hosting device and issue all types of malicious cyber attacks.

While the discovered vulnerabilities are now patched or mitigated, the TLStorm project is not over yet as there are probably many more vulnerable devices. We encourage vendors to make sure that they are using external libraries correctly and not just NanoSSL, but every external library since every external library can be a hidden attack surface. On the users end, this research underlines the need for identification and anomaly detection of network activity for connected devices in order to make sure these devices don't misbehave.