

WHITE PAPER

The S3CUREC4M Project: Vulnerability Research in Modern IP Video Surveillance Technologies

About Nozomi Networks Labs



Nozomi Networks Labs is dedicated to reducing cyber risk for the world's industrial and critical infrastructure organizations. Through its cybersecurity research and collaboration with industry and institutions, it helps defend the operational systems that support everyday life.

The Labs team conducts investigations into industrial device vulnerabilities and, through a responsible disclosure process, contributes to the publication of advisories by recognized authorities.

To help the security community with current threats, they publish timely blogs, research papers and free tools.

The **Threat Intelligence** and **Asset Intelligence** services of Nozomi Networks are supplied by ongoing data generated and curated by the Labs team.

To find out more, and subscribe to updates, visit **[nozominetworks/labs](https://nozominetworks.com/labs)**



Table of Contents

1. Assessing the Security of Modern IP Video Surveillance Technologies	4
1.1 Introduction	4
1.2 Supply Chain Vulnerabilities on Embedded Devices	5
1.3 Assessing Vendor Maturity	6
2. Hardware Analysis and Firmware Extraction Techniques	7
2.1 Introduction	7
2.2 Flash Memories Packages	7
2.3 Memory Dumping Procedures – SOP and WSON	8
2.4 Extracting Firmware from Devices That Don't Support Flashrom	12
2.5 Connecting to a UART Port	15
2.6 JTAG Testing and Analysis	17
3. The Problem of Firmware Observability	21
3.1 Introduction	21
3.2 Transparent Design: Axis Companion Recorder 4CH NVR	21
3.3 Decrypting the Dahua Technology DHI-ASI7213X-T1 Face Recognition Access Controller	23
3.4 From Zero to Debugger: Annke N48PBB NVR	30
4. The Software Attack Surface	34
4.1 Introduction	34
4.2 Management Interfaces	35
4.2.1 Web Management Interface	35
4.2.2 Remote Console	37
4.3 Services Supporting Remote Applications	37
4.3.1 Dahua DVRIP	38
4.4 P2P	38
4.4.1 Reolink P2P Vulnerabilities	39
4.4.2 ThroughTek P2P Vulnerabilities	39
4.4.3 P2P Deployment in Corporate Networks	39
4.5 Cloud Video Surveillance	40
4.6 Discovery Services	40
4.6.1 Hikvision	40
4.6.2 Axis	42
5. Conclusion	43
6. References and Further Reading	44

1. Assessing the Security of Modern IP Video Surveillance Technologies

1.1 Introduction

IP video surveillance systems are likely the most common embedded devices in corporate networks. In 2020, the value of the worldwide video surveillance market surpassed \$45 billion USD and by 2025 is expected to grow to \$75 billion USD.¹ The infrastructure sector—including transportation, city surveillance, public places and utilities—is expected to have the highest growth during that period. One estimate indicates that by the end of 2021, more than a billion IP cameras will be installed worldwide.²

Surveys of internet-accessible video surveillance systems manufactured by the most prominent vendors reveal millions of devices directly reachable through a direct connection. The general public might associate these internet-accessible devices with incidents involving IoT botnets, but video surveillance systems also represent a strategic target for advanced attackers aiming to compromise a specific target.

IP video surveillance systems are typically composed of a set of IP cameras, access control devices, and Network Video Recorders (NVR), where the audio/video stream produced by the cameras is stored, in addition to an application used to access the recordings. Some alternative solutions replace the NVR role with a cloud application for ease of use and better accessibility.

This white paper documents Nozomi Networks' own efforts to study a wide range of video surveillance products in what we are calling the S3CUREC4M Project. It provides security analysts and researchers with a technical framework to help assess the security posture of an IP video surveillance system.

We present a set of analyses that can be performed before deploying the system on a network. We also present a set

of CVE vulnerabilities we have identified and announced as part of this ongoing research project.

Other products solve the remote accessibility issue through a mechanism called Peer-to-Peer (P2P), which should not be confused with traditional peer-to-peer protocols such as BitTorrent. A P2P solution bridges the NVR deployed within a network with remote clients that want to access the audio/video content through the internet.

Easily accessible firmware images are central to assessing the security posture of a device. Unfortunately, some vendors actively obstruct this process, an approach that is harmful to end users and to any necessary analysis. In order to inspect firmware, we first discuss the techniques for obtaining binary code directly from hardware in chapter 2. In some products, firmware is not available for download from the device. Other vendors may distribute encrypted or obfuscated images, though the binary extracted from the device is not encrypted. Either way, firmware dumping is often an essential process to begin the assessment.

Even once a firmware image has been obtained, the executables implementing the services exposed by a device cannot always be freely inspected. Chapter 3 investigates the problem of firmware observability and presents examples of the steps required to analyze the binaries of three different products. We compare a vendor that in our opinion sets the standard for transparency with asset owners, with other vendors that try to block users from inspecting the software that will be running in their networks. In the latter case, we can still successfully unpack the firmware despite the limitations.

Chapter 4 discusses the most common attack surfaces found in IP surveillance systems: management interfaces, services that support remote applications, P2P, cloud video surveillance systems, and discovery services. It also presents some of the vulnerabilities recently discovered by Nozomi Networks Labs, as well as major incidents that concerned those devices.

1.2 Supply Chain Vulnerabilities on Embedded Devices

The firmware image of a typical embedded device is composed of a series of software components packaged together in a single deliverable. These components can either be provided by a commercial software developer, taken from an open-source project, or developed in-house by the vendor.

This process can be recursive, since a component purchased from a commercial software developer is potentially composed of other components originating from third-party developers. More generally, the dependency graph of the software stack that runs on a modern device is quite complex. It's reasonable to assume that very few vendors are aware of the complete software supply chain underpinning their products.

High risk vulnerabilities affecting components at different layers of the stack have emerged recently, requiring a thorough understanding of the software supply chain of an embedded device. This doesn't mean that the risk wasn't there to begin with, but rather that the security community is only now fully understanding the magnitude of the problem.

Recent high-profile vulnerability disclosures have made it clear to security analysts and researchers that deploying networked devices, whose firmware cannot be inspected, is simply not acceptable. Ripple20, for example, affected a TCP/IP stack that was relatively unknown.³

In the case of Ripple20, several vendors weren't even aware that their products were embedded in the vulnerable TCP/IP stack. Other vendors were found to be shipping very

old versions of the vulnerable library, as the contract with the developers had not been renewed. The net result was that end users were left in the dark, unsure if the devices powering their organizations were vulnerable or not.

Network observability is a requirement for this type of situation, as it provides the baseline tools to understand what's happening on the network at a fundamental level. With Ripple20, network traffic analysis makes it possible to identify the devices relying on that specific TCP/IP stack and to warn asset owners if an exploitation attempt is detected.

Firmware image analysis is a complementary approach to network observability, allowing asset owners to further refine their understanding of a device software stack.

Let's suppose that a device is vulnerable to the recent [CVE-2021-3781](#), a flaw in Ghostscript.⁴ Ghostscript is a popular open-source interpreter for PostScript and PDF files and is typically used in the background by software components that perform some higher level tasks. By its very nature, Ghostscript ends up somewhere down in the software stack of an application and its use is not evident to the end user.

Unless a firmware image can be freely inspected, an asset owner has no way of knowing whether this vulnerable software component is deployed in the network, other than testing a proof of concept exploit against a device. This latter approach is obviously subject to the availability of a safe proof of concept, which is rarely the case.

1.3 Assessing Vendor Maturity

For an asset owner, assessing a vendor's maturity is fundamental to understanding the security posture of its products. This is particularly true for embedded devices that expose a significant attack surface at the network level.

A vendor's commitment to releasing security updates in a timely manner is one of the first aspects to consider. This could mean having a rough estimate of how often firmware images are released or determining if the vendor has specific emergency procedures in case a high-profile vulnerability becomes known.

Another key check to perform before picking a product is determining how long the vendor will provide security updates for. Deploying a device with an unclear lifespan of security updates is probably one of the most dangerous situations for an asset owner, as it sets the stage for future incidents.

A further crucial element is the documentation of firmware updates. Each release should be shipped with a description of the patched vulnerabilities as well as documentation on what new features have been inserted. A commonly accepted standard for a software bill of materials is not available yet, but vendors can produce this material in preparation for moving to a format that is accepted industry-wide.

Another very important factor for asset owners is the management of firmware updates for a fleet of devices. Even if a vendor delivers security updates on time, if the process of deploying the new firmware images doesn't scale, the net effect is that devices are still exposed for a long window of time.

Platform hardening is an interesting proxy for how security-focused and dependable a vendor really is. A company that implements modern hardening techniques and is willing to document these efforts for its customers gives a buyer more confidence than a vendor that doesn't apply this defense in depth strategy.

While it's true that no vendor is immune to security vulnerabilities, the type of issues that affect a product is another important measure of its maturity.

The recent hype surrounding supply chain vulnerabilities has made the problem of hidden risks concealed in black box networked products self-evident even to the public. This paper elaborates why these types of products bring unnecessary risks.

2. Hardware Analysis and Firmware Extraction Techniques

2.1 Introduction

As we've mentioned, unencrypted firmware images are not always available. When this occurs, options for analysis are often reduced to a black box interaction with the services exposed to the users. This limitation prevents security researchers from performing a complete and exhaustive assessment and is thus not acceptable. For this reason, extracting firmware directly from a device memory is becoming of paramount importance, as it's often the only way to obtain an unencrypted image.

Hardware analysis is not limited to static firmware extraction. By leveraging debugging/logging ports, such as UART or JTAG, it's possible to interact with a live device. While the former provides some basic interaction capabilities, especially during the boot process, the latter offers a complete hardware debugging environment.

By combining static firmware analysis with dynamic interactions offered by UART or JTAG, researchers can considerably improve their understanding of the attack surface presented by the target.

This chapter provides an overview of commonly used hardware analysis techniques, including how to dump the contents of packages used for PCB-mounted flash memories, the potentiality and usage of the UART port, and JTAG debugging protocol.

2.2 Flash Memories Packages

The first step in the hardware analysis process is to extract the firmware binary. If a vendor is not providing the image, it is necessary to perform a dump directly from the memory of the device. To do so, we first need to identify the flash memory mounted among all the other components soldered on the PCB of the device.

Every electronic component that needs to be installed on a PCB has a hardware interface that allows it to be soldered to the conductive pads of the PCB. There are different types of PCB component packages. This section introduces the three most common, which are usually found in PCB mounted memory devices: SOP, WSON, and BGA.

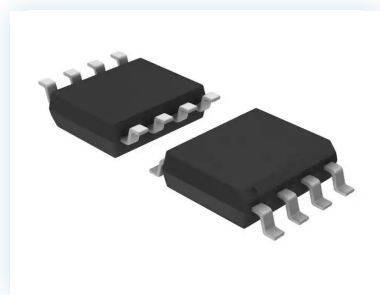


Figure 1a - SOP package.

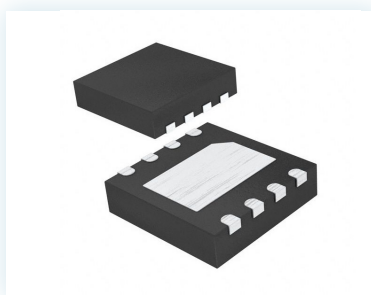


Figure 1b - WSON package.



Figure 1c - BGA package.

The Small Outline Package (SOP) is the most common component package, especially in small embedded devices. Its standard form is a flat rectangular body, with leads extending from two sides. The gull wing shape of the leads allows solid footing during assembly to a PCB. This kind of package facilitates the process of firmware dumping, since the pins can be probed easily by grabbing them with dedicated tools, aptly named grabbers.

The Very-Very-Thin Small-Outline No-Lead (WSON) package is slightly less likely to be found on a PCB. While an SOP package has leads extending from the chip, WSON uses conductive pads. From a hardware analysis perspective, extracting the memory content is more difficult, as the grabbers cannot be used. Rather, some jumpers need to be soldered, or the memory chip may need to be desoldered from the PCB altogether.

The last package that is worth discussion is the Ball Grid Array (BGA). It's a type of surface-mount packaging used for integrated circuits, which can provide more interconnection pins than can be inserted in a dual in-line or flat package. In

contrast to SOP and WSON, a BGA package can use the whole bottom surface of the device, instead of just the perimeter.

The connections of a BGA package are the most difficult to probe, as they aren't accessible from the top of the PCB. Unless they are reachable from the back side of the PCB, the only way to access the package pinout is to desolder the memory from the PCB and insert a socket adapter.

Before we turn to the actual memory dumping process, it's worth highlighting that the memory chips may sometimes need to be removed from the PCB, even if we can reach their pins with grabbers or jumpers. This is due to a possible back propagation of the power through the bus interface.

Depending on how the PCB has been designed, the flash memory and CPU may share the same power line. If this is the case, when the bus interface powers the flash memory, it will also power on the CPU, which will in turn begin communicating with the flash memory. This process effectively blocks any other device from interacting with the chip. Consequently, the component must be desoldered from the PCB for the memory to be read.

2.3 Memory Dumping Procedures – SOP and WSON

As we've mentioned, different flash memory packages require different approaches. In this chapter we'll focus

on SOP and WSON packages, walking through the steps required to read the content from those memories.

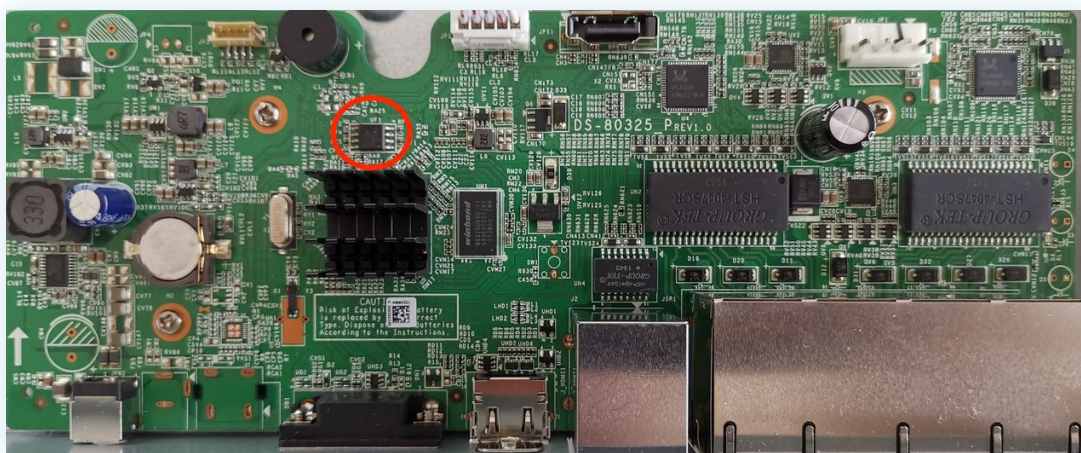


Figure 2 - PCB of the Annke N48PBB. The target SOP flash memory is highlighted with a red circle.

Our first example comes from the analysis of the Annke N48PBB Network Video Recorder, whose PCB is shown in Figure 2. We can spot an SOP-packaged flash memory among the components installed on the PCB, highlighted with a red circle. The label printed on the memory package identifies the vendor name and the model: a **Macronix MXIC MX25L12835F**.

To begin the memory dump process, we start by analyzing the flash memory datasheet to retrieve information about the pinout, supported communication protocol, and operating voltage.

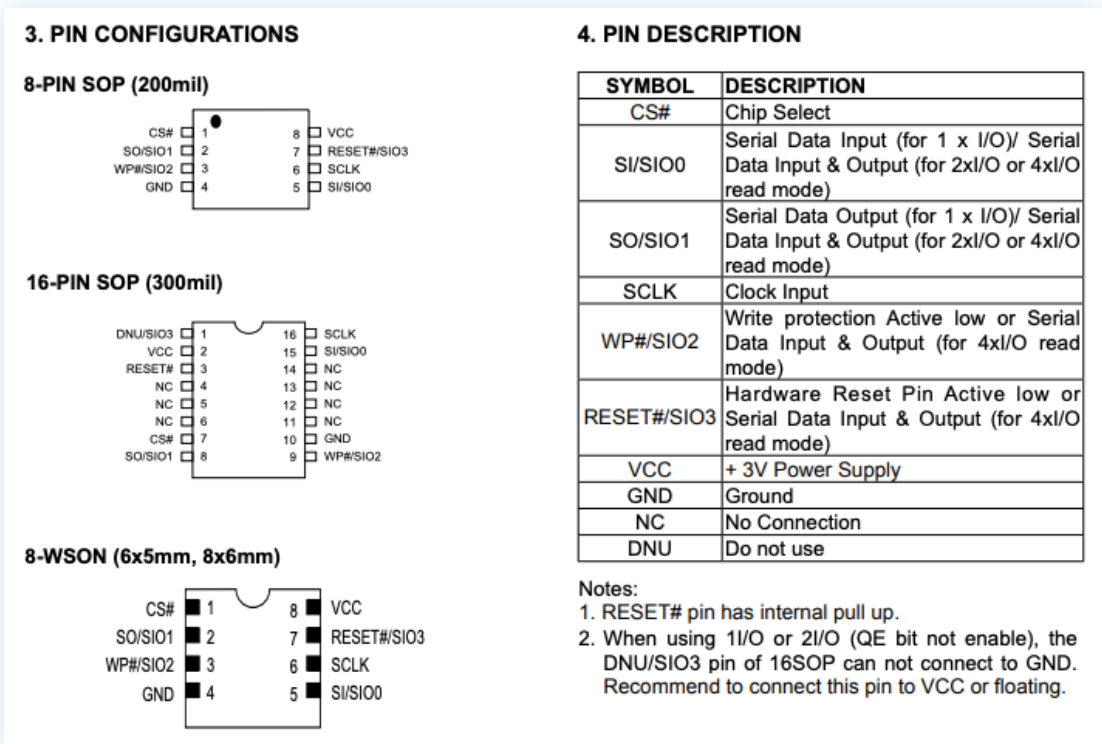


Figure 3 - Datasheet detail of the information required for dumping memory content.

Figure 3 shows part of the Annke N48PBB datasheet, which notes that the memory comes in different packages. The device embeds an 8-PIN SOP, for which a detailed pinout description is provided. The operating voltage value can also be retrieved from the pin description, which in this case is 3V. Finally, the pinout description, in addition to specific information found elsewhere in the datasheet, confirms that the communication protocol supported by this memory is SPI.

Once all the necessary information has been acquired, the next step is to understand which hardware and software components are needed to read the memory content.

One of the most popular softwares used for this type of operation is Flashrom. Flashrom is an open-source utility for identifying, reading, writing, verifying and erasing flash chips. It supports a huge set of flash chips, chipsets, mainboards, PCI and USB devices, and various parallel/serial port-based programmers.

The compatibility of the flash memory must then be checked against a list of supported devices provided in Flashrom documentation.⁵

Macronix	MX25L1005(C)/MX25L1006E	128 SPI	OK	OK	OK	OK	2,700	3,600	Winbond	W25Q64JV	8192 SPI	OK	OK	OK	OK	1,700	1,950
Macronix	MX25L12805D	16384 SPI	OK	OK	OK	OK	2,700	3,600	Winbond	W25Q80.V	1024 SPI	OK	OK	OK	OK	2,700	3,600
Macronix	MX25L12835F/MX25L12845E/MX25L12865E	16384 SPI	OK	OK	OK	OK	2,700	3,600	Winbond	W25Q80BW	1024 SPI	OK	OK	OK	OK	1,700	1,950
Macronix	MX25L1605	2048 SPI	OK	OK	OK	OK	2,700	3,600	Winbond	W25Q80EW	1024 SPI	OK	OK	OK	OK	1,650	1,950
Macronix	MX25L1605A/MX25L1606E/MX25L1608E	2048 SPI	OK	OK	OK	OK	2,700	3,600	Winbond	W25X05	64 SPI	OK	OK	OK	OK	2,300	3,600

Figure 4 - Detail of the list of hardware supported by Flashrom. The **Macronix MXIC MX25L12835F** is highlighted in red, indicating that it is supported by Flashrom.

Figure 4 shows a section of the list of flash chips supported by Flashrom, including the **Macronix MXIC MX25L12835F** memory. This means that the content of our memory can be read using Flashrom, together with a standard bus interface that handles the communications between Flashrom and the memory device. In our second example,

we'll discuss a scenario where the flash chip is not supported by Flashrom.

After gathering the hardware required to dump the memory content, the next step is the setup and the wiring of the bus interface.

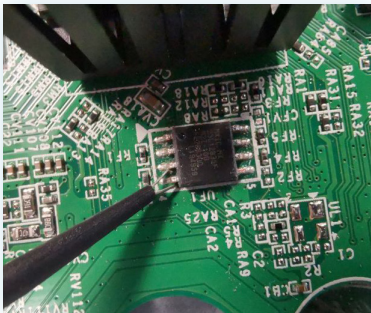


Figure 5a - Grabber positioning.

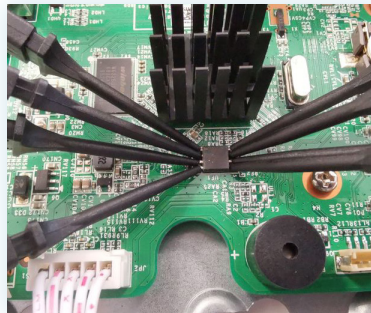


Figure 5b - Every flash chip pin probed with a grabber.

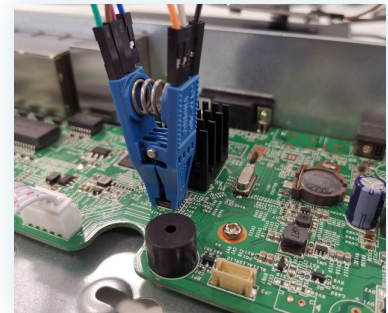


Figure 5c - SOP test clip used instead of grabbers.

There are two main ways to probe the pins of an SOP packaged memory chip: the first uses a set of grabbers that are connected like a clamp to the package pins (Figures 5a

and 5b), while the second leverages an SOP-SOIC test clip to facilitate the probing, avoiding unintended shortcuts between the grabbers (Figure 5c).

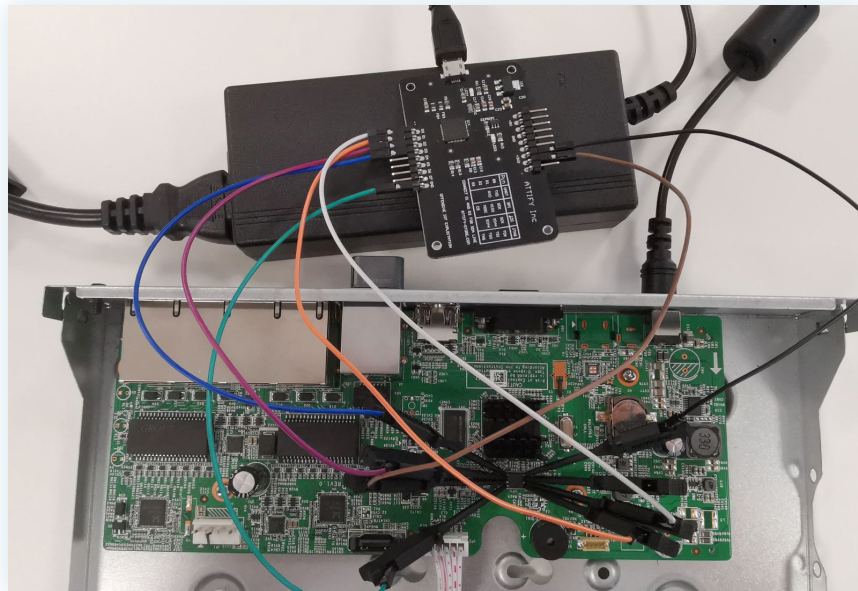


Figure 6 - Wiring setup. An Attify Badge bus interface has been used to manage the SPI communications between the flash memory and the PC running Flashrom.

After the grabbers or the SOIC-SOP clip have been properly connected, a set of jumpers needs to be inserted to connect the grabber to the bus interface. In this setup (Figure 6) an Attify Badge bus interface has been adopted to manage the communications between the flash memory and the PC running Flashrom. Any other bus interface that supports the SPI protocol could also be used.

Figure 3 shows the pinout of the flash chip under analysis. From this schematic we can set up the proper connections according to the pin configuration of the involved bus interface. Notice that the RESET pin of the memory in Figure 6 is left unconnected, as the bus interface does not require the reset signal. Nevertheless, as a good practice, a grabber needs to be connected to avoid unintended contact between the RESET pin and other grabbers.

The last step of this memory dump process consists of the actual reading of the memory content.

Before launching Flashrom, which we're using in this example, it's important to set two parameters: programmer name and chip name.

The programmer name parameter depends on which bus interface is being used. In this case, Attify Badge is based on an FTDI chip communicating with the memory through SPI protocol. In the Flashrom manual, the programmer name for this kind of bus interfaces corresponds to **ft2232_spi:type=232H**.

The chip name is the model of the flash memory, which can be found in the Flashrom list of supported hardware. Figure 4 identifies the name of the chip we are reading from: **MX25L12835F/MX25L12845E/MX25L12865E**.

The option that enables the setting of the programmer name is **-p**, while the chip name is **-c**.

The complete Flashrom command will then be:

```
flashrom -p ft2232_spi:type=232H -c MX25L12835F/  
MX25L12845E/MX25L12865E -r image.bin
```

The **-r** option, on the other hand, tells Flashrom to perform a reading operation. The output of this command will eventually be a dump of the entirety of the flash chip's content, which will be saved in file **image.bin**.

2.4 Extracting Firmware from Devices That Don't Support Flashrom

In some hardware, the flash memory may have a WSON package that is not supported by Flashrom. To provide a complete overview of the techniques that allows security

engineers to dump firmware images from flash memories, we present a second analysis where this is the case.

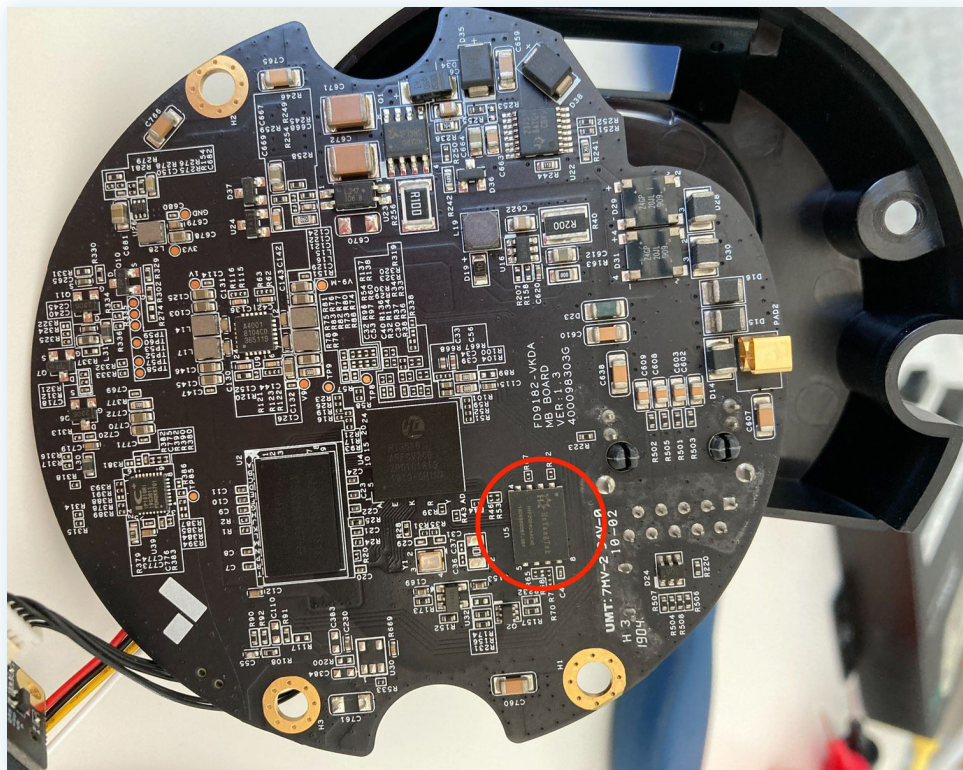


Figure 7 - PCB overview of the Verkada D40 camera. A red circle highlights the flash memory.

The target device is a Verkada D40 camera, whose PCB is shown in Figure 7. The red circle highlights the flash memory, which in this case is a HeYangTek HYF2GQ4UAACAE.

As in the previous example, the first step involves retrieving useful information regarding the chip. Unfortunately, this time very little information could be obtained, only voltage and package type.

Furthermore, this chip was not found on the list of hardware supported by Flashrom, making it unlikely that Flashrom can be used to extract the content. It should be noted that some officially unsupported flash memories are effectively a re-branding of a compliant model, in which case Flashrom will operate as expected.

After soldering some jumpers to the pad of the flash memory, a reading test is performed, using the common pinout for SPI WSON flash chips. Unfortunately, Flashrom does not recognize the flash. At this point, only a dedicated

programmer would be able to read the memory content. We can find a programmer that supports the content by searching the internet - BeeProg2C by Elneec, with a WSON-8 adapter.

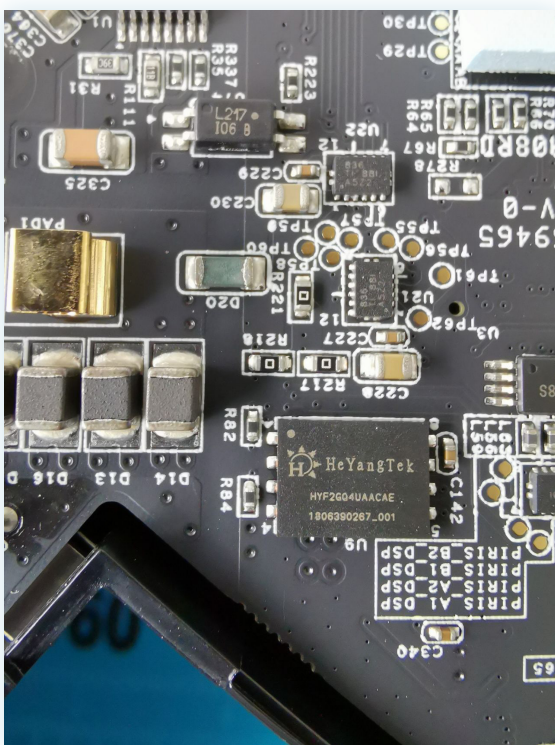


Figure 8a - The HeYangTek HYF2GQ4UAACAE mounted on the Verkada D40 PCB.

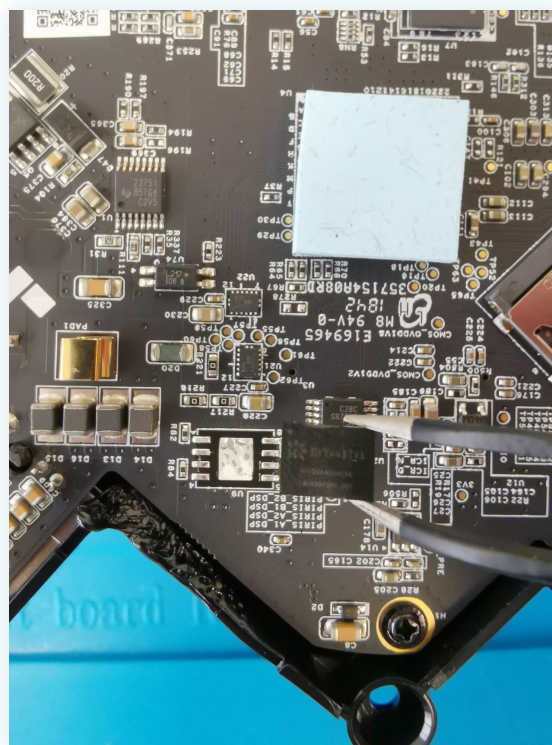


Figure 8b - The flash memory desoldered from the PCB.

To be able to read the memory content, the flash chip needs to be desoldered from the Verkada D40 PCB with a hot air desolder tool.

Figures 8a and 8b show the HeYangTek HYF2GQ4UAACAE flash chip before and after being desoldered from the PCB. After removing the chip, it is usually a good practice to

clean up both the flash chip pads and the PCB pads from where the memory was desoldered.

A small amount of tin should also be put on the pads of the memory chip, in order to ease the contact with the socket adapter of the programmer.



Figure 9a - The BeeProg2C with the socket adapter for WSON-8 memories.

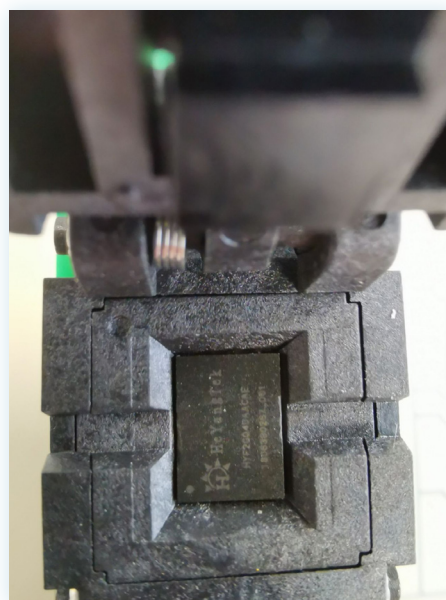


Figure 9b - The HeYangTek HYF2GQ4UAACAE flash chip positioned in the socket adapter.

The BeeProg2C programmer (shown in Figure 9a) is very easy to use. The memory chip only needs to be inserted

into a socket adapter (Figure 9b) and the programmer connected to a PC running Windows.

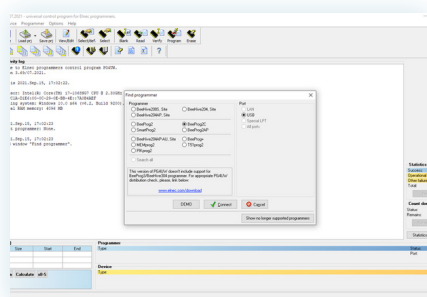


Figure 10a - Programmer selection.

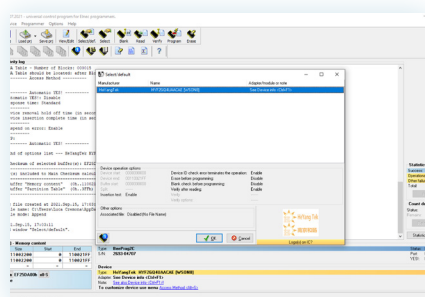


Figure 10b - Flash chip selection.

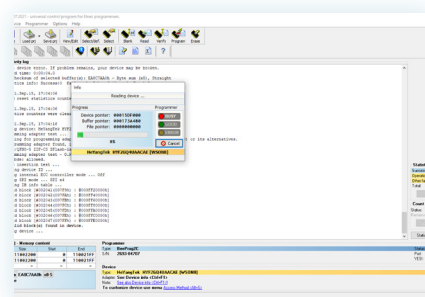


Figure 10c - Dumping procedure.

At the startup of PG4UW software, the model of the programmer and the flash chip must be specified (Figures 10a and 10b). Once this simple setup has been completed, the content of the memory can be extracted (Figure 10c). This process requires several minutes to complete, after which the content of the memory will be stored in a buffer that can be saved to any file specified by the user.

In the bottom right corner of PG4UW UI there is a small table with the statistics of the dumping process, which reports the number of reading successes and failures. Failures can happen during the dumping process; if they do, the reading process will need to be restarted.

2.5 Connecting to a UART Port

Low level softwares, such as bootloaders, don't usually implement drivers for sophisticated communication devices. Rather, they often employ simple interfaces that provide a very basic communication functionality between the user and the device.

The most common low level communication interface is the Universal Asynchronous Receiver-Transmitter (UART). A UART is a hardware device capable of establishing asynchronous serial communications, where the data format and the transmission speed are configurable. Data bits are sent one by one, from least to the most significant, and are organized in frames interleaved by a start and stop bit.

From an electrical point of view, the voltage level is handled according to two main systems, RS-232 (12 volts) and RS-485 (5 volts). Sometimes the UART system can be implemented with a dedicated Integrated Circuit (IC), but most of the time, it is embedded within the main microprocessor. An evolution of UART, called USART, can also handle synchronous transmissions.

Due to the simplicity of its communication interface, UART is widely used in applications with hardware or software constraints. By default, it's the only communication infrastructure for which U-Boot, one of the most common bootloaders for IoT systems, has a standard support.

UART is leveraged for all communications between the device and a possibly connected host PC, before the actual firmware or kernel starts. It can be used not only to send output from the board to the host PC, but also to receive commands from the PC, allowing auditors to interact with the boot process.

From a physical point of view, a UART connection is comprised of four terminals: Ground (GND), Voltage in (VCC), Receive (RX) and Transmit (TX). In many devices, VCC is not needed, as UART hardware is already powered on by the device itself.

When looking for a serial port in a PCB, the search should be focused on four-pins connectors. However, the connector is often not mounted on the PCB and only four aligned holes are left on the board.

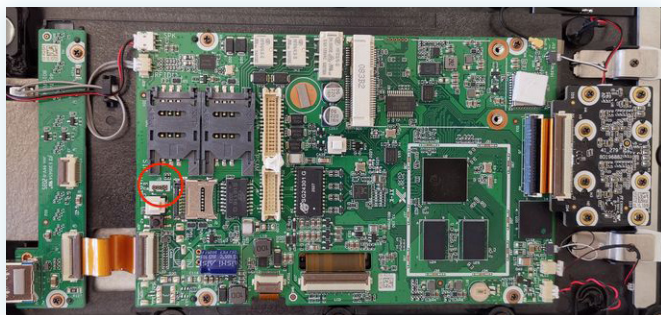


Figure 11a - PCB of the Dahua DHI-ASI7213X-T1 thermal camera. A red circle highlights the UART terminals.

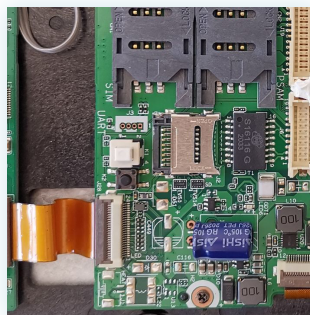


Figure 11b - No connectors are mounted, requiring one to be soldered.

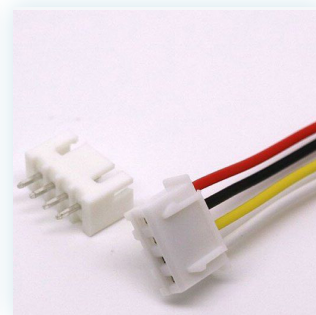


Figure 5c - JST cable and connector.

The following example describes the connection to the UART port and the reading of the boot information for a Dahua DHI-ASI7213X-T1 thermal camera.

Figure 11a shows the Dahua thermal camera PCB, in which UART terminals can be confirmed by labels printed on the PCB. No connectors are soldered to the UART terminal holes.

There are two techniques that can be used to connect the thermal camera to the PC through the UART. One option is to solder a JST connector to the PCB and use JST cables as a bridge between the PCB and the bus interface. Another method is to solder four jumpers directly on the terminal holes.

After setting up the connections for both the PCB and the bus interface side, we can now open a console and connect to the UART. In this setup, an Attify Badge was used as a bus interface, while the common **screen** application was the terminal for connecting to the UART port.

After understanding which command is reserved for stopping U-Boot from booting the kernel, ***** in the case of Dahua DHI-ASI7213X-T1, it's finally possible to open a **screen** and switch on the camera.

```
1  System startup
2  allowed version 00000000, major=0, minor=0
3  device support otp.
4  Otp version is 0x00000000, Flash version is 0x00000000
5  Otp version is 0x00000000, Flash version is 0x00000000
6  UB00T_commonSwRsaVerify run successfully!
7
8  U-Boot 2016.11-svn8097 (May 09 2020 - 02:27:46 +0800)hi3519av100
9
10 dhboot #
```

As shown in the listing above, we then get a shell to interact with U-Boot information. In addition to being a good source of information about both the device hardware and the kernel, the UART can also be used to gather access to the

underlying operating system. For example, by modifying specific environmental variables, it might be possible to obtain a shell after the kernel has completed its startup.

2.6 JTAG Testing and Analysis

The Joint Test Action Group (JTAG) is an industry standard for verifying designs and testing PCBs after manufacture. It was originally defined in the 80's to tackle the problem of testing integrated circuits and communication busses as they were becoming faster and miniaturized. Since the old analog probes were no longer effective for circuit testing purposes, a new methodology had to be invented.

The key concept of JTAG is to move the testing infrastructure from the outside to the inside of an integrated circuit. The Intel i486 DX2 was the first microprocessor embedding a complete JTAG-compliant scan chain.

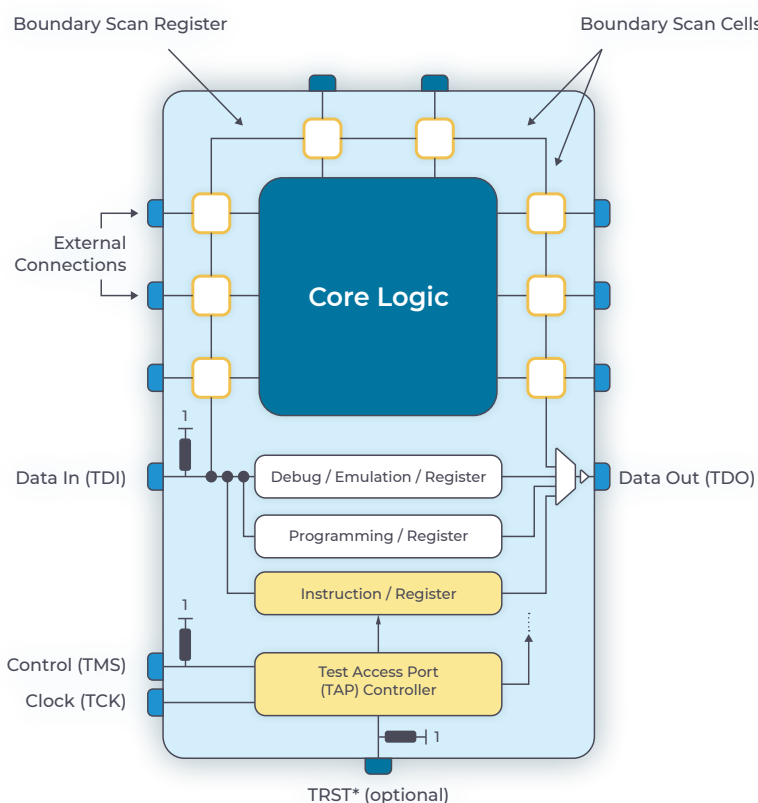


Figure 12 - The JTAG scan chain.

A complete JTAG scan chain implemented within a System on Chip (SoC) is shown in Figure 12. The JTAG standard exposes five pins: Test Data In (TDI), Test Data Out (TDO), Test Mode Select (TMS), Test Clock (TCK), Ground (GND), and an

optional Test Reset (TRST). Given this information, during the analysis of the PCB, connectors with either four, five or six pins should be the targets of an auditor.

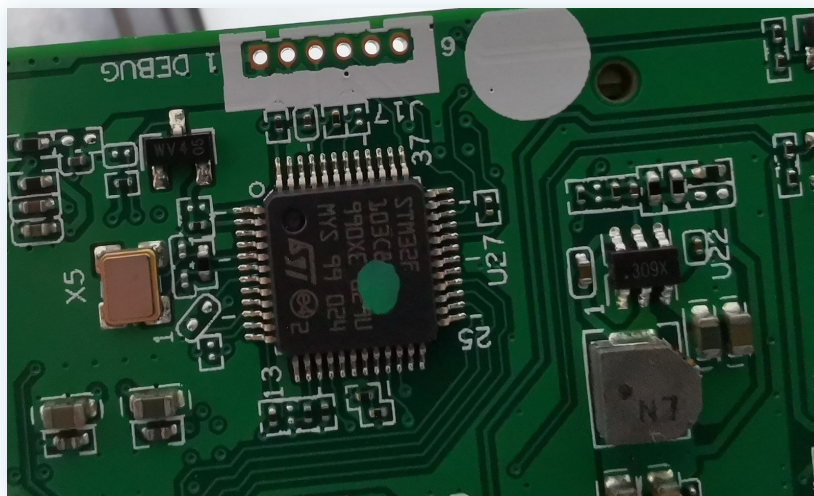


Figure 13 - A six-pin JTAG connector. It is the debug port for the STM32 in Figure 14.

Figure 13 shows a detail of the back side of the Dahua DHI-ASI7213X-T1 PCB. The label on the top left of the picture identifies the six-pins above the STM32 as the interface of a debug port. The fact that they are very close to the

STM32 microcontroller suggests that this is the debug port for the STM32 itself. To understand which role each JTAG terminal has, we need to consult the schematics of the microcontroller and match the pinout using a multimeter.

Figure 8. STM32F103xx performance line LQFP48 pinout

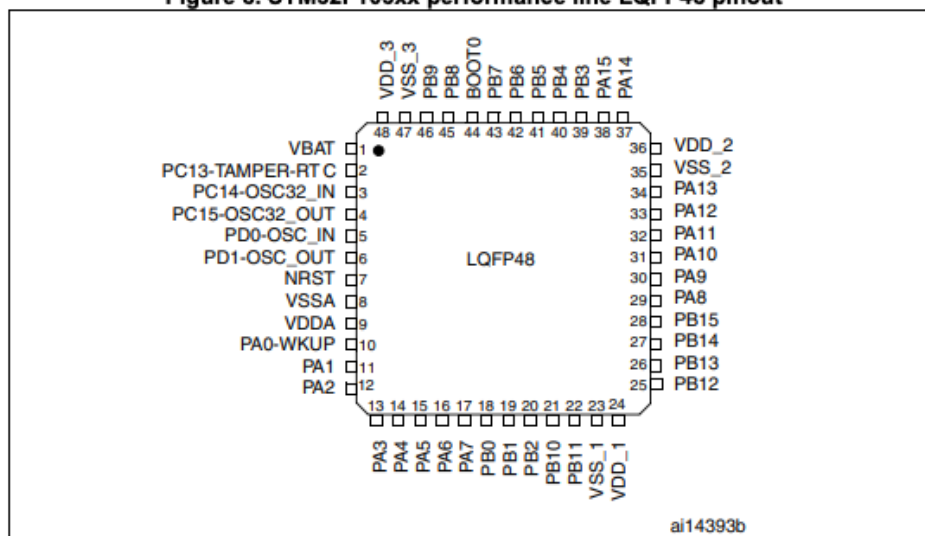


Figure 14 - The pinout schematic of the STM32F103C8 microcontroller.

In the datasheet of the STM32F103C8 microcontroller, the following mapping of the JTAG is described:

- TDI → PA15
- TDO → PB3
- TMS → PA13
- TCK → PA14
- TRST → PB4

With the use of a multimeter set in continuity mode, we verified that the order from left to right is: GND, PB3, PB4, PA13, PA14, PA15 (shown in Figure 14). Having gathered all the information needed for the hardware setup, we can now focus on software. We only need two applications to have a functional debugging environment, OpenOCD and GDB.

OpenOCD is an open-source project that allows in-system programming, boundary scan testing and debugging for multiple MIPS and ARM systems. GDB can be run simultaneously with OpenOCD, to better understand all the instructions the CPU is executing and to investigate the content of registers and the flash memory.

To kick off a debugging session, OpenOCD needs to be started with this command:

```
sudo ./openocd -s ../tcl -f stm32.cfg
```

If the interface of choice is Attify Badge, `stm32.cfg` can be obtained directly from Attify.⁶

The terminal output should resemble the following:

```

1  Open On-Chip Debugger 0.11.0-rc2
2  Licensed under GNU GPL v2
3  For bug reports, read
4    http://openocd.org/doc/doxygen/bugs.html
5  Info : auto-selecting first available session transport "jtag". To override
6  use 'transport select <transport>'.
7  adapter speed: 15000 kHz
8  Info : Listening on port 6666 for tcl connections
9  Info : Listening on port 4444 for telnet connections
10 Info : Hardware version: 9.30
11 Info : VTarget = 2.605 V
12 Info : clock speed 15000 kHz
13 Info : JTAG tap: stm32.cpu tap/device found: 0x00000001 (mfg: 0x000
    (<invalid>), part: 0x0000, ver: 0x0)
14 Info : starting gdb server for stm32.cpu on 3333
15 Info : Listening on port 3333 for gdb connections
```

From the output of OpenOCD, it we can see that there are three ports on listening:

- TCP port 6666: for tcl connections
- TCP port 4444: for telnet connections
- TCP port 3333: for GDB connections

To test if the setup is working properly, it is now possible to connect to the telnet server and trigger the reset procedure:

```
1 Telnet localhost 4444
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 Open On-Chip Debugger
6 >reset
7 JTAG tap: stm32.cpu tap/device found: 0x00000001 (mfg: 0x000
  (<invalid>), part: 0x0000, ver: 0x0)
```

The actual debugging session can then be started with the following command:

```
gdb extended-remote :3333
```

From now on, GDB can be used as it would in a normal debug session.

The procedure described above is a general approach to hardware debugging with OpenOCD and GDB, which can be applied to any CPU or SoC and leverage any bus interface.

However, for STM-8 and STM-32 microprocessors, there is a dedicated probe for hardware debugging and programming called ST-Link. It can be used together with STMCube, the IDE for STM microprocessors, and provides a very good and intuitive GUI, enabling better and easier debugging sessions.

3. The Problem of Firmware Observability

3.1 Introduction

Firmware observability is the ability of a customer or a third party to freely inspect the binaries that implement the services exposed by a given device. This analysis can take place statically, for instance by using a disassembler to examine specific executables, or at runtime while the target system is executing.

In recent years, an emerging trend has involved many vendors actively obfuscating or encrypting firmware. Their goal is to block any type of analysis other than a pure black box interaction. In some cases, vendors even describe these efforts as security driven.

Our experience tells us instead that the opposite is true; it's actually dangerous for any organization of a given complexity to deploy networked products that cannot be easily analyzed.

As there are many approaches to firmware distribution, ranging from accessible to deliberately obfuscated, we discuss three examples from different vendors and their potential consequences for the security posture of an organization.

3.2 Transparent Design: Axis Companion Recorder 4CH NVR

Axis Communications is a video surveillance company that adopts an open policy towards firmware inspection. The Companion Recorder 4CH is a typical network video

recorder (NVR) which is managed through a web interface. SSH access can be enabled through the management interface and the device can be inspected at runtime.

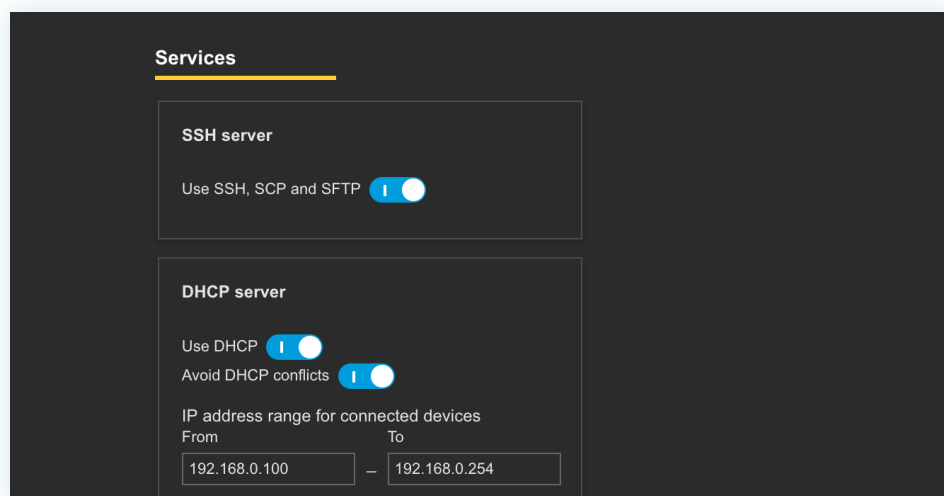


Figure 15 - Enabling SSH access.

3.3 Decrypting the Dahua Technology DHI-ASI7213X-T1 Face Recognition Access Controller

DHI-ASI7213X-T1 is a face recognition access controller which, among the details, can detect the temperature of the person looking into the device. A web management interface is available to configure the access controller. SSH

access can be enabled, but to our surprise, the credentials set for the web interface don't apply to the remote shell. Our understanding is that SSH access is available only to Dahua support, should the need arise.

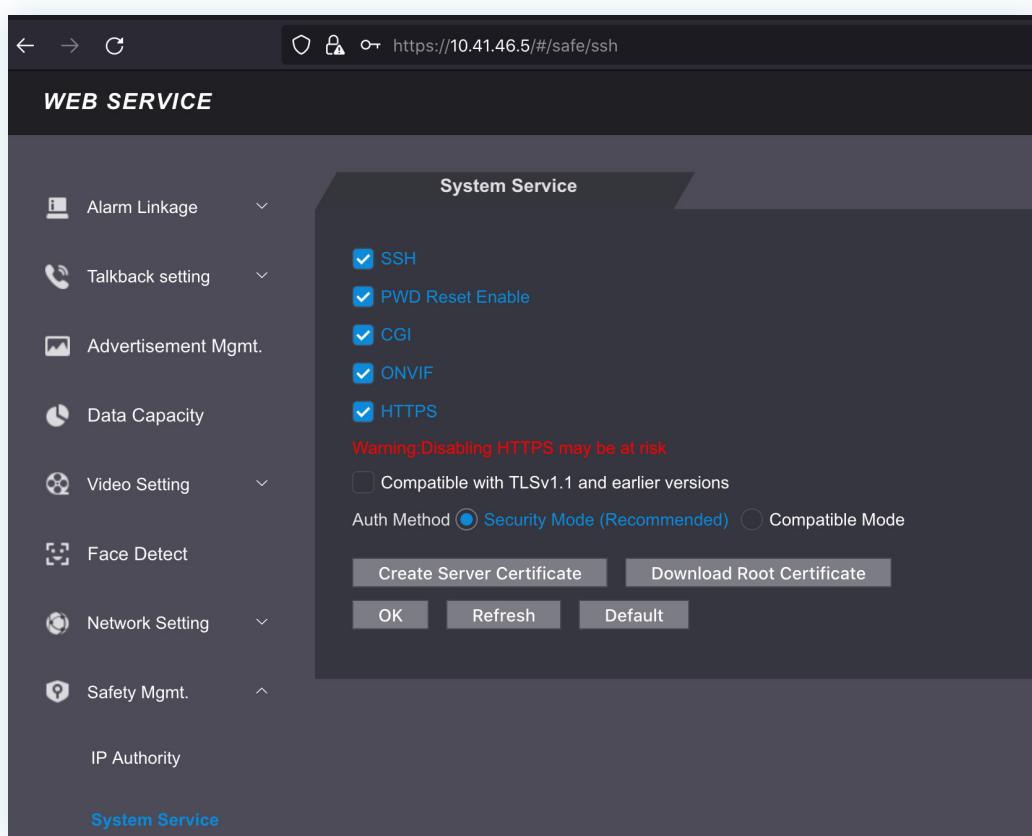


Figure 18 - The credentials set for the web interface of the DHI-ASI7213X-T1 do not apply to the remote shell.

Firmware can be downloaded from the vendor website, but as you unpack the binary with Binwalk, you'll notice that the process does not proceed as expected. The tool successfully extracts a series of ulmage files, but the content of most of

these binaries is encrypted with a proprietary scheme. In particular, the kernel and the partition images containing the final executables are not accessible.

```

noname:extracted $ file *
Install: JSON data
check.img: u-boot legacy uImage, check, Linux/ARM, Firmware Image (Not compressed), 1440 bytes, Sun Dec 13 08:47:11 2020, Load Address: 0x00000000, Entry Point: 0x00000000, Header CRC: 0xFC7B446C, Data CRC: 0x1FE7EB44
data-x.squashfs.img: u-boot legacy uImage, data, Linux/ARM, Standalone Program (gzip), 14895288 bytes, Sun Dec 13 08:43:03 2020, Load Address: 0x49B00000, Entry Point: 0x4FF00000, Header CRC: 0xF760F0FF, Data CRC: 0x3F6AE03B
dhboot-min.bin.img: u-boot legacy uImage, boot, Linux/ARM, Firmware Image (Not compressed), 133024 bytes, Sun Dec 13 08:42:34 2020, Load Address: 0x00000000, Entry Point: 0x00100000, Header CRC: 0x8AD09337, Data CRC: 0x2CBC44AF
dhboot.bin.img: u-boot legacy uImage, boot, Linux/ARM, Firmware Image (Not compressed), 272752 bytes, Sun Dec 13 08:42:34 2020, Load Address: 0x00200000, Entry Point: 0x00300000, Header CRC: 0x49B4989F, Data CRC: 0x00ACFF02
firmware-x.squashfs.img: u-boot legacy uImage, firmware, Linux/ARM, Standalone Program (gzip), 128518328 bytes, Sun Dec 13 08:46:59 2020, Load Address: 0x04400000, Entry Point: 0x70C00000, Header CRC: 0x99A00FCF, Data CRC: 0x983E6553
hwids: empty
kernel.img: u-boot legacy uImage, kernel, Linux/ARM, Firmware Image (Not compressed), 4135760 bytes, Sun Dec 13 08:42:34 2020, Load Address: 0x04600000, Entry Point: 0x05600000, Header CRC: 0xE18C4638, Data CRC: 0x61901D40
partition-x.cramfs.img: u-boot legacy uImage, partition, Linux/ARM, Standalone Program (gzip), 6328 bytes, Sun Dec 13 08:42:34 2020, Load Address: 0x05600000, Entry Point: 0x05700000, Header CRC: 0xE583D778, Data CRC: 0xAC9CB36C
pd-x.squashfs.img: u-boot legacy uImage, pd, Linux/ARM, Standalone Program (gzip), 100536 bytes, Sun Dec 13 08:46:06 2020, Load Address: 0x03600000, Entry Point: 0x04600000, Header CRC: 0x9B91D82A, Data CRC: 0xDD8B4555
ramfs-x.squashfs.img: u-boot legacy uImage, ramfs, Linux/ARM, Standalone Program (gzip), 41031864 bytes, Sun Dec 13 08:45:56 2020, Load Address: 0x05700000, Entry Point: 0x0D75023A1, Data CRC: 0x3BA37A17
sign.img: data
web-x.squashfs.img: u-boot legacy uImage, web, Linux/ARM, Standalone Program (gzip), 4884664 bytes, Sun Dec 13 08:42:42 2020, Load Address: 0x0D700000, Entry Point: 0x10700000, Header CRC: 0x0936A715, Data CRC: 0xD09AFA58

```

Figure 19 - Encrypted ulmage files extracted from Dahua firmware.

We searched publicly available documentation to understand if the vendor documented their approach to firmware security. We eventually found a product security white paper which explicitly states that firmware is encrypted “to prevent reverse attacks by hackers.”⁸ We

believe that the opposite is actually true, namely that the lack of an accessible firmware image harms asset owners more than malicious actors. We then set off to analyze the encryption scheme implemented in this product.

4.3.2.6 Anti-Reverse

Firmware is one of the important assets of the device. In order to prevent reverse attacks by hackers, Dahua has designed a firmware encryption scheme to ensure that the firmware remains encrypted during the data transfer process. The basic principle is as follows:

- Create a security key based on KDF technology and encrypt the firmware data;
- When the device upgrades the firmware, write it to Flash in encrypted form;
- During the device startup process, the Flash partition data is decrypted and loaded.

17



Figure 20 - A Dahua white paper states that firmware is encrypted to prevent hacking.

From the artifacts previously unpacked, the bootloader stored in file **dhboot.bin.img** was found not be encrypted. We then reverse engineered a considerable part of this

binary and eventually reached the function responsible for decrypting the kernel.

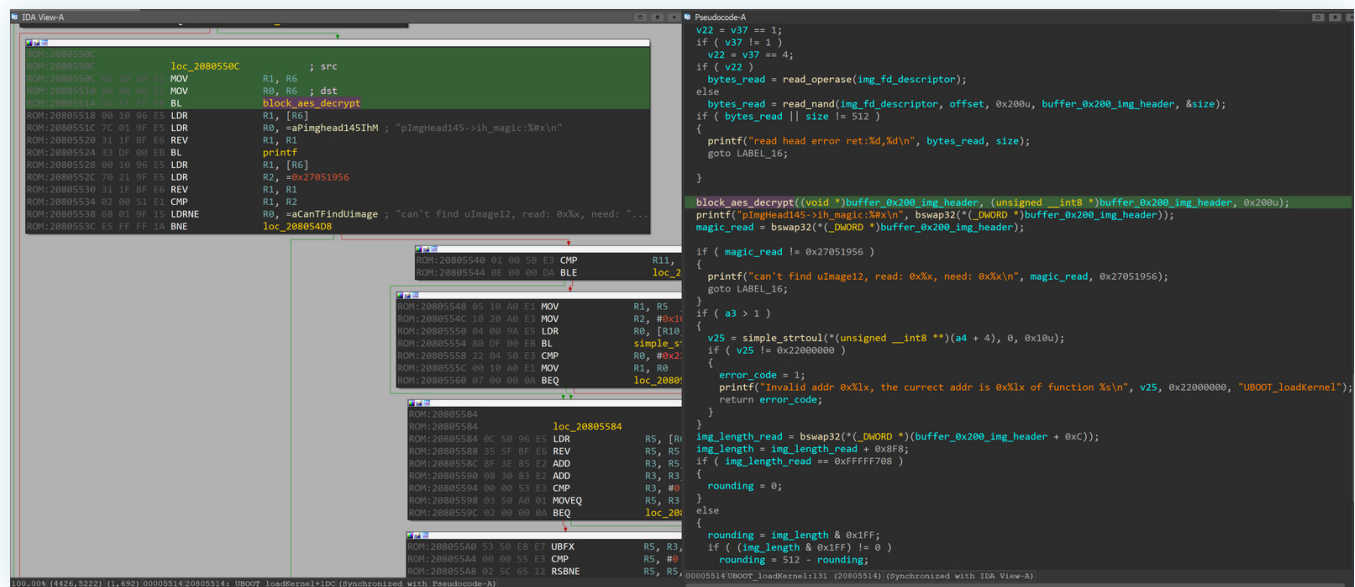


Figure 21 - The function responsible for decrypting the kernel.

The decryption scheme is based on AES-ECB with a key derived from the SHA256 of a hardcoded key (Figure 22).

```
int __fastcall firmware_get_key(int out_key)
{
    int string_length; // r0
    char *v3; // r3
    int i; // r2
    char v5; // t1
    unsigned __int8 key_buffer[32]; // [sp+0h] [bp-198h] BYREF
    char SHA256Context[112]; // [sp+20h] [bp-178h] BYREF
    char data[264]; // [sp+90h] [bp-108h] BYREF

    memset(key_buffer, 0, sizeof(key_buffer));
    memcpy(data, &word_208481FB, 256);
    string_length = strlen("hi3519av100_emmc");
    memcpy(data, "hi3519av100_emmc", string_length);
    sha256_init(SHA256Context);
    sha256_update(SHA256Context, data, 256);
    sha256_final(SHA256Context, key_buffer);

    v3 = SHA256Context;
    for ( i = 0; i != 16; ++i )
    {
        v5 = *--v3;
        *(_BYTE *)(out_key + i) = key_buffer[i] ^ v5;
    }
    return 0;
}
```

Figure 22 - Key derivation.

Since the decryption functions contained some customizations on top of the block cipher and we didn't want to waste more time reimplementing the whole scheme, we opted for an emulation-based approach and let the original code perform the decryption for us.

As the addresses to which the bootloader expects to be mapped are compatible with userspace memory layout, we wrote a loader that would take the bootloader binary as the input and map it at the correct address.

```
char* map_address_space(char* fname, int fd, size_t buffer_length, size_t img_size) {

    void* rom_address = (void*) 0x20800000;
    printf("[*] mapping input file %s at 0x%x\n", fname, (unsigned int) rom_address);
    char* res = map_buffer(rom_address, fd, buffer_length);

    void* dunno_address = (void*) 0x04530000;
    printf("[*] mapping dunno at 0x%x\n", (unsigned int) dunno_address);
    res = map_empty_buffer(dunno_address, 0x1000);

    void* bss_address = (void*) 0x2085D000;
    printf("[*] mapping bss at 0x%x\n", (unsigned int) bss_address);
    res = map_empty_buffer(bss_address, 0x60000);

    void* elf_address = (void*) 0x22000000;
    printf("[*] mapping elf at 0x%x\n", (unsigned int) elf_address);
    res = map_empty_buffer(elf_address, img_size);

    return res;
}
```

Figure 23 - Bootloader binary mapping.

We then defined the function pointers for the decryption routine we intended to emulate and set the corresponding

address within the mapped executable code.

```
typedef int (*some_stupid_copy)(unsigned char *dst, unsigned char *src, int length);
typedef int (*copy_key_stuck_in_the_middle)(unsigned int dst, char *src_firmware_key, unsigned int key_length);
typedef char* (*set_result_to)(char *result, unsigned int zero, int size);

typedef int (*block_aes_decrypt)(void *dst, unsigned char *src, unsigned int size);
typedef int (*firmware_get_key)(char* out_key);
typedef int (*set_key_factors)(char* firmware_key, int int_16, unsigned int int_0);
typedef unsigned int (*u_boot_encrypt)(char *src, unsigned int src_size, char* dst, int dst_size, unsigned char firmware_key_delta, int action, int* out_size);

typedef int (*aes_expand_encryption_key)(unsigned char *key, int key_length, int *expanded_key);
typedef int (*aes_expand_decryption_key)(unsigned char *key, int key_length, int *expanded_key);
typedef int (*aes_decrypt)(int, char *, unsigned int, char *, char *);
typedef int (*aes_encrypt)(int, char *, int, unsigned int, char *);

block_aes_decrypt fpBlock_aes_decrypt = (block_aes_decrypt) 0x20805284;
firmware_get_key fpFirmware_get_key = (firmware_get_key) 0x208051E4;
set_key_factors fpSet_key_factors = (set_key_factors) 0x208232D4;
u_boot_encrypt fpU_boot_encrypt = (u_boot_encrypt) 0x20823338;

aes_expand_encryption_key fpAes_expand_encryption_key = (aes_expand_encryption_key) 0x20821F18;
aes_expand_decryption_key fpAes_expand_decryption_key = (aes_expand_decryption_key) 0x20822314;
aes_decrypt fpAes_decrypt = (aes_decrypt) 0x20822E7C;
aes_encrypt fpAes_encrypt = (aes_encrypt) 0x20822D50;
```

Figure 24 - Pointing encryption code to mapped locations.

Once the loader was ready, we ran it and decrypted the kernel successfully, as shown in Figure 25.

```

debian-arm:loader $ ./loader u-boot-padded.bin ./trimmed/kernel_trimmed.bin
[*] reading file ./trimmed/kernel_trimmed.bin
[*] rounding: 96
    size 4123552
[*] resulting size 4123648
[*] opening file u-boot-padded.bin
    size 770336
[*] mapping input file u-boot-padded.bin at 0x20800000
[*] mapping dunno at 0x4530000
    map result 0x4530000
[*] mapping bss at 0x2085d000
    map result 0x2085d000
[*] mapping elf at 0x22000000
    map result 0x22000000
[*] decrypting kernel
[*] firmware_get_key ok
[*] firmware key:
00 | 93 20 8F 3D A9 7C 53 4D 44 6B 95 E6 C9 4A 1F 6F | . .|.SMDk...J.o
[*] derived key before:
00 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
10 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
20 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
40 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
set key factors ok
[*] derived key after:
00 | 93 20 8F 3D A9 7C 53 4D 44 6B 95 E6 C9 4A 1F 6F | . .|.SMDk...J.o
10 | E6 76 EC F9 04 BF F4 2F FF AF 03 52 99 DA 0E EB | .v..../.R...
20 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
40 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
[*] ciphertext snippet:
00 | 19 32 1F C9 8C B8 B9 A5 60 59 9C 32 6C 7E 69 ED | .2.....`Y.2l~i.
10 | AD 06 39 78 0C C3 68 C7 5B 8F 42 B4 AD 35 BE 0E | ..9x..h.[.B..5..
20 | EA C1 00 3D 99 BD 0A 97 EA 51 E8 FE 00 67 C9 E4 | ...=.....Q...g...
30 | FE 0C BB DE 0A D4 70 97 8E F4 47 F8 D3 1F D3 54 | .....p...G....T
[*] aes key (0x0x20883c74):
00 | 93 20 8F 3D A9 7C 53 4D 44 6B 95 E6 C9 4A 1F 6F | . .|.SMDk...J.o
10 | E6 76 EC F9 04 BF F4 2F FF AF 03 52 99 DA 0E EB | .v..../.R...
u-boot encrypt ok
[*] cleartext snippet:
00 | 27 05 19 56 7C E6 9C 91 5E BA 2F 43 00 3E D9 F0 | '..V|...^./C.>..
10 | 22 00 80 00 22 00 80 00 98 67 F5 7F 05 02 02 00 | "..."...g.....
20 | 4C 69 6E 75 78 2D 34 2E 39 2E 33 37 00 00 00 00 | Linux-4.9.37....
30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
[*] kernel decrypted!

```

Figure 25 - Successful kernel decryption through emulation.

We tried the same procedure with the encrypted files containing the device partitions, but the process was unsuccessful. Since the kernel is executed right after the bootloader, we continued our analysis by reverse engineering the newly decrypted code.

As we determined at the end of the decryption process, the kernel of this device is a Linux-4.9.37 with some

customizations. Its sheer size and complexity are such that we had to resort to some heuristics to find the functions in charge of decrypting the remaining images.

We looked for the constants used by AES as a starting point. Once we could clearly define the boundaries of the AES implementation, we looped through all the functions that reference the block cipher algorithm.


```
kernel:C0828774 ; _DWORD aes_U0[256]
kernel:C0828774 50 A7 F4 51+ aes_U0 DCD 0x51F4A750, 0x7E416553, 0x1A17A4C3, 0x3A275E96, 0x3BAB68CB
kernel:C0828774 53 65 41 7E+ ; DATA XREF: aes_expand_decryption_key+4C to
kernel:C0828774 C3 A4 17 1A+ ; aes_expand_decryption_key+5C to ...
kernel:C0828774 96 5E 27 3A+ DCD 0x1F9D45F1, 0xACFA58AB, 0x4BE30393, 0x2030FA55, 0xAD766DF6
kernel:C0828774 CB 6
kernel:C0828774 F1 4
kernel:C0828774 AB 5
kernel:C0828774 93 6 Up o aes_expand_decryption_key+4C LDR R0, =(aes_U0 - 0xC06B726C)
kernel:C0828774 55 F Up o aes_expand_decryption_key+5C ADD R0, PC, R0; aes_U0
kernel:C0828774 F6 6 Up o kernel:off_C06B73F8 DCD aes_U0 - 0xC06B726C
kernel:C0828774 91 7 Up o sub_C06B7850+44 LDR R10, =(aes_U0 - 0xC06B78AC)
kernel:C0828774 25 4 Up o sub_C06B7850+54 ADD R10, PC, R10; aes_U0
kernel:C0828774 FC D Up o kernel:off_C06B7C50 DCD aes_U0 - 0xC06B78AC
kernel:C0828774 80 4
kernel:C0828774 8F A
kernel:C0828774 49 5
kernel:C0828774 67 1
```

Line 1 of 6

OK Cancel Search Help

Figure 26 - AES constants and references as found in kernel.

We eventually located the same decryption routines that we initially found in the bootloader. The caveat, though, was that the key derivation function was not in proximity of the decryption code. With some further analyses we also

identified the key derivation function and realized that the encryption scheme for the partition is precisely the same as the one used for the kernel. What differs is a simple positional parameter.

```
loc_C051B270
05 20 A0 E1 MOV R2, R5
06 30 A0 E1 MOV R3, R6
1C 00 18 E5 LDR R0, [R8, #-0x1C]
A6 A4 F9 EB BL sg_copy_to_buffer
05 10 A0 E1 MOV R1, R5 ; src
00 E0 A0 E3 MOV LR, #0
28 30 9D E5 LDR R3, [SP, #0x80+var_58]
34 20 9D E5 LDR R2, [SP, #0x80+var_4C]
00 E0 8D E5 STR LR, [SP, #0x80+action] ; action
02 C0 43 E0 SUB R12, R3, R2
30 00 9D E5 LDR R0, [SP, #0x80+dst] ; dst
11 30 A0 E3 MOV R3, #0x11 ; key_offset
06 20 A0 E1 MOV R2, R6 ; size
04 C0 8D E5 STR R12, [SP, #0x80+out_size] ; out_size
9A EE FF EB BL decrypt_partition
06 30 A0 E1 MOV R3, R6 ; size
24 10 18 E5 LDR R1, [R8, #-0x24] ; int
08 50 A0 E1 MOV R5, R0
30 20 9D E5 LDR R2, [SP, #0x80+dst] ; decrypted_buffer
1C 00 18 E5 LDR R0, [R8, #-0x1C] ; int
8D A4 F9 EB BL outer_sg_copy_buffer
00 00 55 E3 CMP R5, #0
07 FF FF 0A BEQ loc_C051AEEC
```

```
528 v90 = md[0x13A];
529 v91 = v90 == 0;
530 if ( v90 )
531 v91 = src == 0;
532 dst_1 = md[0x13A];
533 if ( !v91 )
534 {
535 if ( size )
536 {
537 _memzero(src, size);
538 _memzero(dst_1, size);
539 }
540 v85 = *(brq - 9);
541 LABEL_162:
542 sg_copy_to_buffer(*(brq - 7), v85, src, size);
543 res = decrypt_partition(dst_1, src, size, 0x11u, 0, v106 - v111);
544 outer_sg_copy_buffer(*(brq - 7), *(brq - 9), dst_1, size);
545 if ( !res )
546 goto blk_end_request;
547 LABEL_163:
548 print_error(&mmc_decry_failed, v92);
549 goto blk_end_request;
550 }
551 }
552 else
553 {
```

Figure 27 - Partitions decryption function as found in kernel.

This finding meant that we could simply tweak our existing decryption tool and use it to decrypt the

remainder of the firmware.

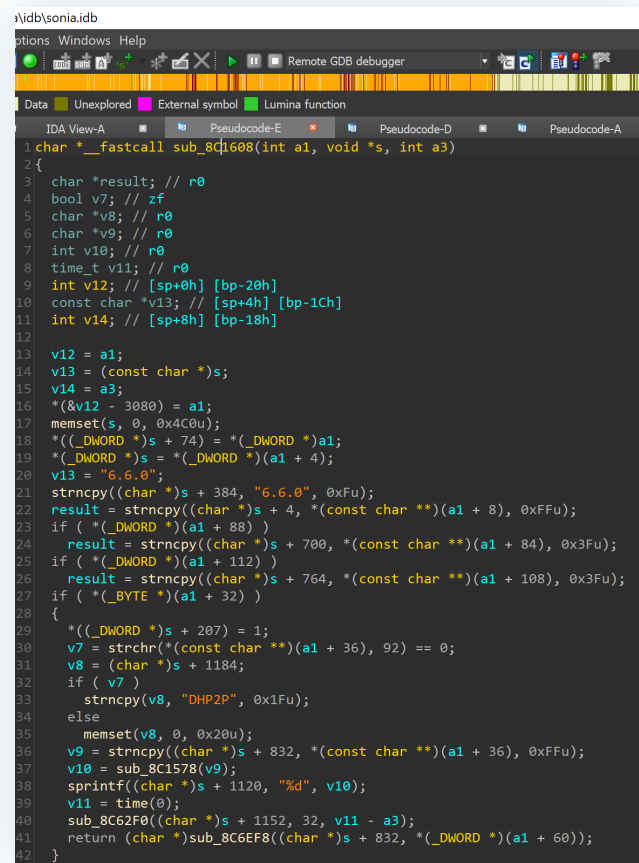

```

debian-arm:loader $ ./loader u-boot-padded.bin ./trimmed/romfs-x.squashfs.img_trimmed
[*] reading file ./trimmed/romfs-x.squashfs.img_trimmed
[*] rounding: 328
    size 40667320
[*] resulting size 40667648
[*] opening file u-boot-padded.bin
    size 770336
[*] mapping input file u-boot-padded.bin at 0x20800000
[*] mapping dunno at 0x4530000
    map result 0x4530000
[*] mapping bss at 0x2085d000
    map result 0x2085d000
[*] mapping elf at 0x22000000
    map result 0x22000000
[*] decrypting partition
[*] firmware_get_key ok
[*] firmware key:
00 | 93 20 8F 3D A9 7C 53 4D 44 6B 95 E6 C9 4A 1F 6F | . . .ISMDk...J.o
[*] derived key before:
00 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
10 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
20 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
40 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
[P] set key factors ok
[*] derived key after:
00 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
10 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
20 | 93 20 8F 3D A9 7C 53 4D 44 6B 95 E6 C9 4A 1F 6F | . . .ISMDk...J.o
30 | 5E 2E 79 54 70 70 34 F3 EC 30 F5 88 9F 53 3B CE | ^.yTpp4..0...S;.
40 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
[*] ciphertext snippet:
00 | 4E B9 6D DF 4A 85 44 8B CB A8 E8 D2 8D 00 5A F0 | N.m.J.D.....Z.
10 | FE CF BC E9 A7 4D 2C 9B F7 A7 89 75 48 92 D0 43 | .....M.....uH..C
20 | 51 F5 B8 C5 8D D6 5F 64 93 80 79 4C EE E4 33 08 | Q.....d..yL..3.
30 | 25 DB EB 4F B1 9E 40 38 4F 73 71 E6 73 27 E0 DC | %..0...@80sq.s'..
[P] u-boot encrypt ok
[*] cleartext snippet:
00 | 68 73 71 73 36 02 00 00 17 AF C7 5E 00 00 02 00 | hsq6.....^....
10 | 3D 00 00 00 04 00 11 00 C0 04 02 00 04 00 00 00 | =.....
20 | 51 1B F4 10 00 00 00 00 FE 7B 6C 02 00 00 00 00 | Q.....{l.....
30 | F6 7B 6C 02 00 00 00 00 FF FF FF FF FF FF FF FF | .{l.....
[*] partition decrypted!

```

Figure 28 - Finishing the decryption

We could finally run binwalk on the decrypted squashfs image and access the firmware partitions statically. With



```

1 char *__fastcall sub_8C1608(int a1, void *s, int a3)
2 {
3     char *result; // r0
4     bool v7; // zf
5     char *v8; // r0
6     char *v9; // r0
7     int v10; // r0
8     time_t v11; // r0
9     int v12; // [sp+0h] [bp-20h]
10    const char *v13; // [sp+4h] [bp-1Ch]
11    int v14; // [sp+8h] [bp-18h]
12
13    v12 = a1;
14    v13 = (const char *)s;
15    v14 = a3;
16    *(&v12 - 3080) = a1;
17    memset(s, 0, 0x4C0u);
18    *((_DWORD *)s + 74) = *((_DWORD *)a1);
19    *((_DWORD *)s) = *((_DWORD *)a1 + 4);
20    v13 = "6.6.0";
21    strncpy((char *)s + 384, "6.6.0", 0xFu);
22    result = strncpy((char *)s + 4, (const char **)(a1 + 8), 0xFFu);
23    if ( *((_DWORD *)a1 + 88) )
24        result = strncpy((char *)s + 700, (const char **)(a1 + 84), 0x3Fu);
25    if ( *((_DWORD *)a1 + 112) )
26        result = strncpy((char *)s + 764, (const char **)(a1 + 108), 0x3Fu);
27    if ( *((_BYTE *)a1 + 32) )
28    {
29        *((_DWORD *)s + 207) = 1;
30        v7 = strchr((const char **)(a1 + 36), 92) == 0;
31        v8 = (char *)s + 1184;
32        if ( v7 )
33            strncpy(v8, "DHP2P", 0x1Fu);
34        else
35            memset(v8, 0, 0x20u);
36        v9 = strncpy((char *)s + 832, (const char **)(a1 + 36), 0xFFu);
37        v10 = sub_8C1578(v9);
38        sprintf((char *)s + 1120, "%d", v10);
39        v11 = time(0);
40        sub_8C62F0((char *)s + 1152, 32, v11 - a3);
41        return (char *)sub_8C6EF8((char *)s + 832, *((_DWORD *)a1 + 60));
42    }

```

Figure 29 - A decompiled function from executable "sonia."

full access to the executable binaries, the usual analysis can now be performed.

3.4 From Zero to Debugger: Annke N48PBB NVR

The Annke N48PBB is an NVR capable of showing and recording the footage of up to eight Power over Ethernet (PoE) IP security cameras. Like the Dahua access controller, it exposes a web management interface which provides the

possibility of enabling SSH access. In this case, the credentials of the admin account are accepted by the device; however, it is only possible to obtain access to a restricted shell which allows the execution of a limited set of commands.

```
Enter 'help' for a list of DVR/NVR system commands.

# help
Support Commands:
GetAnrCfgInfo          GetAnrProcess          GetAnrRecordList
ShowIpcAbility         accessDvrSwitch        channelPlayback
clearDisksMode         ctrlArchDebug          decStat
disableHB              disableHik264          dspStatus
dvrLogInfo             dt                     enableHB
enableHik264           enableWatchdog          errputClose
errputOpen             get3GMode              getCMS
getCycleReboot         getDbgCtrl             getHardInfo
getIp                  getLastErrorInfo       getPlayTestCtrl
getPort                getServerInfo          guiChkCfg
guiEnterMenuCount      guiPrtScr              guiStatus
help                   helpu                  i2cRead
megaDspConfig          miscCmd                netstat
outputClose            outputOpen             partRecDetails
ping                   printPart              pthreadInfo
recorderChanInfo       recorderFileInfo       recorderFileKeyFrame
recorderHdIdle         recorderMediaInfo      recorderPAllocFile
recorderParam          recorderSegExtraInfo   recorderStatus
sendATCom              set3GPrint             set3GEnable
searchInfo             setGateway             setIp
setlang                setMtu                 setoutputmode
setPrint               show8107coreUseInfo    showCurPlayChanFileInfo
showDeviceTemp         showIpcMemInfo         showNetIpcmInfo
showNetLinksInfo       showPlayChanStatus    showPlayClipFile
showPlayScreenInfo     showPlayStatus         showPlayTime
showPreviewInfo        showShareSvcInfo      showSpareWorkStatus
showTagSysInfo         showUserInfo           showpu
t1                     t2                     transcodeResStatus
getDateInfo            dmesg                  help
debug
#
```

Figure 30 - Annke N48PBB restricted shell.

None of the commands allowed out-of-the-box deeper access to the OS internals. Some known bypasses were attempted but were unsuccessful. Additionally, at the time of the analysis, no firmware was available to download from the Annke website.

In order to gain complete access to the Annke NVR, we decided to dump the firmware directly from the flash

memory via SPI. As a matter of fact, the flash memory in use by the device proved to be a **Macronix MXIC MX25L12835F**, which has pins big enough for micro grabbers to connect steadily and is supported by the well-known flash memory tool Flashrom. Section 2.3 of this document describes the technique used.

This allowed us to obtain a firmware image that binwalk could successfully analyze. Figure 31 shows the content of

the CramFS filesystem extracted from the firmware image.

```

└─# ls -al
total 15244
drwxrwxrwx 2 root 232 4096 Sep 20 12:22 .
drwxr-xr-x 3 root root 4096 Sep 20 12:22 ..
-rwxr-xr-x 1 root 232 133120 Jan 1 1970 de.tar
-rwxr-xr-x 1 root 232 133120 Jan 1 1970 en.tar
-rwxr-xr-x 1 root 232 143360 Jan 1 1970 es.tar
-rwxr-xr-x 1 root 232 143360 Jan 1 1970 fr.tar
-rw-r--r-- 1 root 232 2047304 Jan 1 1970 gui_res.tar.xz
-rwxr-xr-x 1 root 232 133120 Jan 1 1970 it.tar
-rwxr-xr-x 1 root 232 153600 Jan 1 1970 ja.tar
-rwxr-xr-x 1 root 232 1256 Jan 1 1970 new_10.bin
-rwxr-xr-x 1 root 232 859992 Jan 1 1970 player.zip
-rwxr-xr-x 1 root 232 143360 Jan 1 1970 pt.tar
-rwxr-xr-x 1 root 232 184320 Jan 1 1970 ru.tar
-rwxr-xr-x 1 root 232 3696 Jan 1 1970 start.sh
-rw-r--r-- 1 root 232 5361488 Jan 1 1970 sys_app.tar.xz
-rwxr-xr-x 1 root 232 3182232 Jan 1 1970 uImage
-rwxr-xr-x 1 root 232 2417496 Jan 1 1970 WebComponents.exe
-rwxr-xr-x 1 root 232 543968 Jan 1 1970 webs.tar.xz

```

Figure 31 - Listing of the CramFS filesystem content

In order to modify the firmware and remove the protected shell, an injection of ad-hoc lines in the `start.sh` script seemed to be the best option. We determined through publicly available information from the community and confirmed with tests that it is executed at each device booting process.

Similar to Dahua, this file, as well as many others on the firmware, was encrypted by Annke to protect against at-rest modifications. However, Annke is known to unofficially

resell Hikvision devices under their name as an OEM,⁹ and further research revealed that most Hikvision devices used to adopt an identical encryption key and mechanism, which are known to the community. The `hikpack` tool (developed by the user “montecrypto”¹⁰) successfully decrypted and re-encrypted the files. Option `-t k51` was used, as it was the latest Hikvision NVR supported by this tool, and it experimentally proved to be correct.

```

└─# cat -v start_dec.sh
#!/bin/sh
#rm /bin/psh
sdbg=$(/usr/bin/awk -F 'sdbg=' '{print substr($2,1,1)}' /proc/cmdline)
who=$(/usr/bin/awk -F 'who=' '{print $2}' /proc/cmdline|awk '{print $1}')
data=$(/usr/bin/awk -F 'data=' '{print $2}' /proc/cmdline|awk '{print $1}')
nfsdir=$(/usr/bin/awk -F 'nfsdir=' '{print $2}' /proc/cmdline|awk '{print $1}')
serverip=$(/usr/bin/awk -F: '{print $2}' /proc/cmdline)
gdb=$(/usr/bin/awk -F 'gdb=' '{print $2}' /proc/cmdline|awk '{print $1}')
echo "sdbg:$sdbg serverip:$serverip"

```

Figure 32 - Portions of the decrypted start.sh startup script.

The script was modified in order to replace the original `/bin/psh` with another executable script file called `busybox ash`. Then, it was re-encrypted by using the `hikpack` tool.

Unfortunately, the results were still insufficient: in addition to encrypting files, further analyses determined that Annke protects their integrity by comparing the computed MD5

hash of each file in the flash memory at firmware boot time with the one stored in the `new_10.bin` file, which is also encrypted. Fortunately, the encryption scheme was the same one used to encrypt the `start.sh` file; thus, by replicating the steps we used on the `start.sh` file on the `new_10.bin` file, it was possible to update the MD5 hash of the startup script in order to pass the validation process.

```

$ hexdump -C new_10_dec.bin
00000000  15 03 15 20 00 00 00 00  00 00 00 00 00 00 00 00  |... ..|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
00000020  00 00 00 00 07 00 00 00  72 6f 6f 74 0a 00 00 00  |.....root....|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
000000a0  00 00 00 00 00 00 00 00  4d 6f 6e 20 41 70 72 20  |.....Mon Apr |
000000b0  31 32 20 32 30 3a 33 37  3a 34 35 20 32 30 32 31  |12 20:37:45 2021|
000000c0  0a 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
00000120  00 00 00 00 00 00 00 00  75 49 6d 61 67 65 00 00  |.....uImage..|
00000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
00000140  00 00 00 00 00 00 00 00  ff 1a b3 d4 ef 54 ee 29  |.....T.)|
00000150  20 ca c6 6e 38 df f1 21  00 00 00 00 00 00 00 00  |..n8..!.....|
00000160  00 00 00 00 00 00 00 00  73 74 61 72 74 2e 73 68  |.....start.sh|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
00000180  00 00 00 00 00 00 00 00  86 55 cd db 55 aa 57 27  |.....U..U.W'|
00000190  8d c5 3d c0 87 7f 9a ae  00 00 00 00 00 00 00 00  |..=.....|
000001a0  00 00 00 00 00 00 00 00  57 65 62 43 6f 6d 70 6f  |.....WebCompo|
000001b0  6e 65 6e 74 73 2e 65 78  65 00 00 00 00 00 00 00  |nents.exe.....|

```

Figure 33 - Portions of the decrypted `new_10.bin` file.

In order to obtain a newly working firmware, the CramFS filesystem was rebuilt by invoking the `mkfs.cramfs` command and rewritten on the firmware image, replacing

the previous filesystem. After reflashing the device memory with the new firmware image, we obtained unrestricted access to the device.

```

$ ssh 10.41.46.18
The authenticity of host '10.41.46.18 (10.41.46.18)' can't be established.
RSA key fingerprint is SHA256:DBjW1ARZJu2KeYKBdyknb71LpR6+a9hLgghBjmioMc8.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.41.46.18' (RSA) to the list of known hosts.
root@10.41.46.18's password:
[root@dvrds /root] # id
uid=0(root) gid=0(root) groups=0(root)
[root@dvrds /root] # hostname
dvrds

```

Figure 34 - Unrestricted SSH access to the device.

Inspections of the running processes revealed that most functionalities of the device are handled by the `/home/app/exec/master` binary. However, the first attempt to debug the binary by attaching a statically-compiled gdbserver was ineffective. This is due to the presence of a watchdog, monitoring the status of the process and triggering a device reboot in case of needs.

Luckily, the results of the `start.sh` startup script contained a debugging branch in an if-else statement with the exact command necessary to disable it. By invoking the command highlighted in Figure 35, it was finally possible to attach a gdbserver and fully debug the executable.

```

fi

if [ "$sdbg" == "g" ];then
    telnetd &
    echo C > /proc/hkvs/wdtinfo
else
    sleep 1
    echo "start to start hik_app"
    /home/app/exec/netOpenProc &
    chmod 777 ./GPIProc

```

Figure 35 - Watchdog-disabling code in the decrypted start.sh startup script.

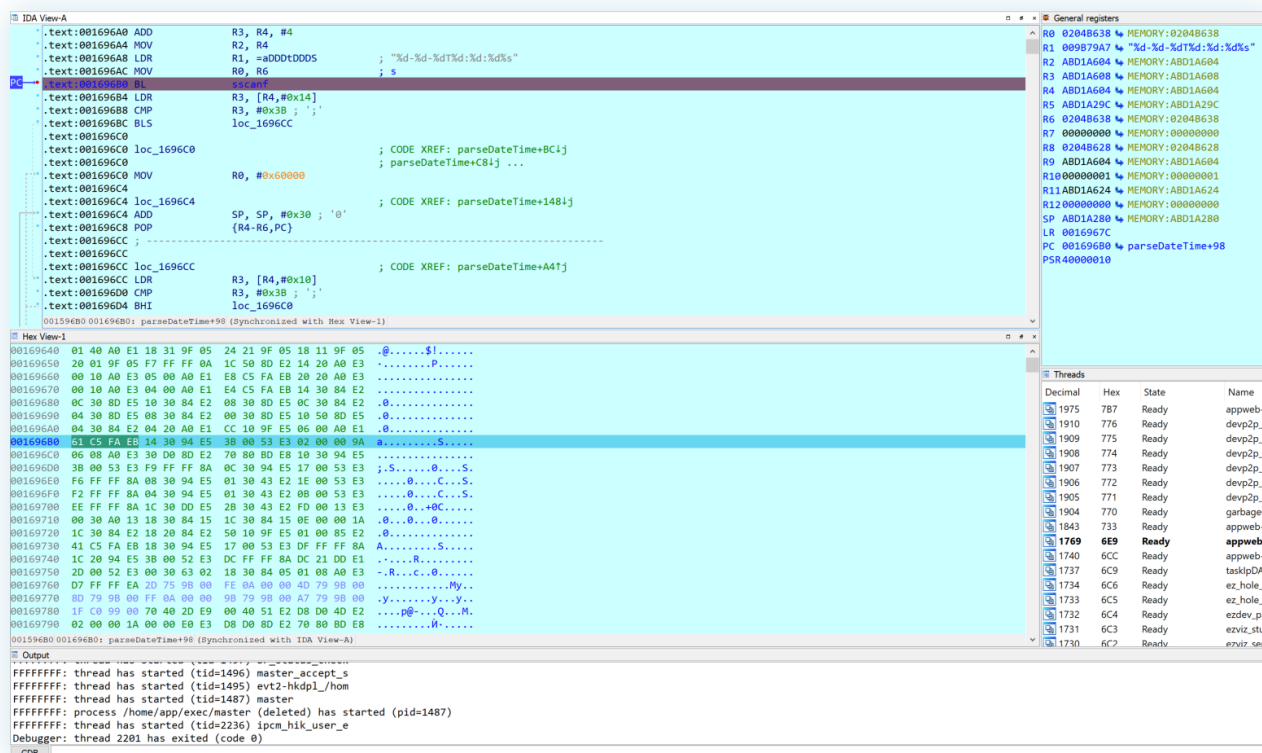


Figure 36 - Debugging session of master binary in IDA Pro.

4. The Software Attack Surface

4.1 Introduction

After getting full access to a device's firmware and deploying it to a test network, it's finally time to decompose its attack surface and assess its security posture. Having a remote debugger onboard the device considerably speeds up the process of analyzing each specific target service. Nevertheless, when debugging a target is not an option, it's still possible to gather a thorough understanding of the device through static analysis and black box interaction.

An IP surveillance system has a minimum set of three logical services which are required to perform its basic tasks.

The first concerns management interfaces, which for on-premises installations are typically exposed through a web application running on the devices. Cloud solutions are instead managed through the provider's SaaS (software as a service) platform interface.

The second service determines how the audio/video stream is transferred from the producers of the data, the cameras, to the device that provides the storage, the network video recorder. In the case of a cloud surveillance system, the camera has an internal memory that holds the real-time recording until it's later transferred to the SaaS platform.

The third type of basic service concerns how recordings are eventually accessed. This could be achieved through an application interacting with an NVR through a specific protocol on the same network. Some software stacks instead provide the recordings via the same web application used to manage the device.

An additional feature is the possibility of streaming the audio/video content through the internet with P2P (peer-to-peer), which in this context should not be confused with the concept of peer-to-peer as implemented in protocols such as BitTorrent. This feature was initially found on many consumer products but has lately been making its way into corporate solutions.

Cloud-based systems, as expected, allow users to replay the recordings through a SaaS platform, which is naturally accessed through the internet.

In addition to these services, which are integral to the basic functioning of a surveillance system, there are additional features that some vendors might decide to implement, for instance to simplify the management of a fleet of devices. Discovery services fall into this category, as they expose always-on network reachable code which, if exploited, could allow an attacker to compromise devices at scale.

To assess the security posture of a surveillance system, the first challenge is to understand which boundaries between services can be the subject of an in-depth analysis and which should not. An understanding of which parts of a video surveillance system are implemented on the devices and which functionalities rely instead on the cloud platform managed by a provider is needed.

Of course, an auditor won't have any constraints while testing a device functionality, such as the login of a management interface. If a specific feature relies instead on a cloud service, the possible ramifications of the activities must be carefully assessed before beginning the process.

This chapter focuses on the most common software attack surfaces found in IP surveillance systems: management interfaces, services that support remote applications, P2P, cloud video surveillance systems, and discovery services.

Let's suppose that an auditor is testing a video surveillance system with an on-premises NVR that automatically backs up the recordings to a cloud platform managed by the vendor. One of the tests might involve setting a description for a specific recording to particular values, with the goal of testing the on-device web interface for the presence of Cross-Site Scripting (XSS) vulnerabilities. If this malicious description is then reflected in the cloud backup system,

the auditor might accidentally trigger a vulnerability on a vendor-managed system.

If not agreed upon and discussed in advance with the vendor, this sort of test involving cloud platforms might fall somewhere between being tolerated to being considered unacceptable. If the goal of an asset owner is to understand the overall posture of a system including its cloud features, prior discussions should be held with the vendor. Any company taking security seriously is in fact already

performing internal audits periodically, and likely already has a process in place to inform customers about its security process. After establishing a contact, an asset owner can then decide if the information provided is enough or can negotiate further assessments with the vendor.

Let's now dive into specific instances of the aforementioned attack surfaces and discuss some security issues that emerge.

4.2 Management Interfaces

4.2.1 Web Management Interface

Like the majority of modern IoT devices, among the management services exposed by a generic IP video surveillance system, the web interface is undoubtedly one of the most commonly available and utilized. This is largely because it can be interacted with from virtually any client, without needing to install vendor-specific software.

The presence of a web channel, on the other hand, means that design and development of even apparently simple devices must take web vulnerabilities into account. This applies to the entire range of web vulnerabilities that IT administrators have come to know over the years on complex enterprise web applications.

Now more than ever, the risk of a web application attack cannot be ignored. According to the Verizon's 2021 *Data Breach Investigation Report*, web application attacks are the second most utilized pattern in breaches or incidents against companies.¹¹ Barracuda's *The State of Application Security in 2021* report reinforces this concept, stating that, on average, organizations were successfully breached twice in the period from mid-2020 to mid-2021 as a direct result of an application vulnerability.¹²

The architecture of a generic web application running on an embedded device can be as complex as that of an enterprise web application. Figure 37 shows a sample scheme which outlines its tiers.

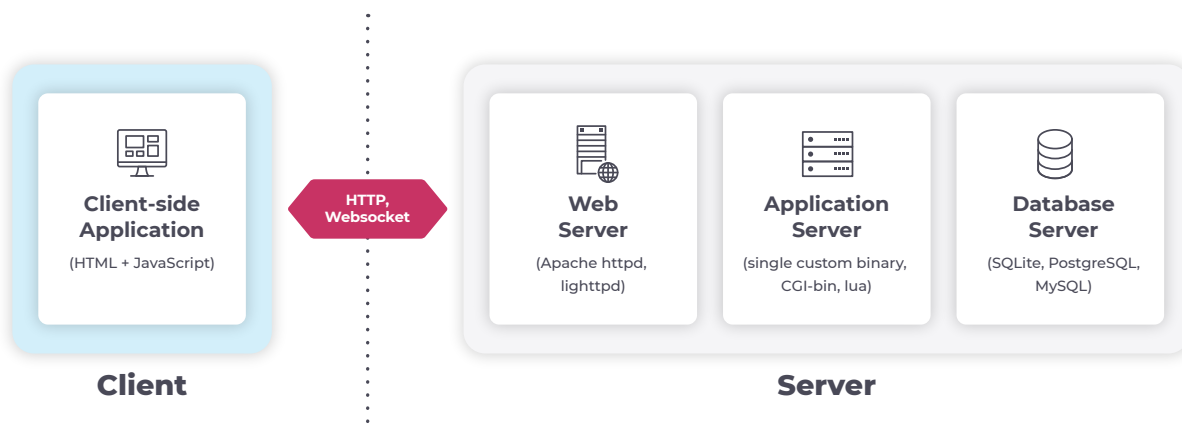


Figure 37 - Sample web application architecture of an embedded device.

All these tiers need to be thoroughly inspected for vulnerabilities, as web application flaws can hide in any of them. These are only few examples of the possible security bugs which may be found in an embedded web application:

- A DOM-based XSS vulnerability in the client-side application, due to insufficient escaping of user input by the JavaScript code before being rendered in the page;
- A Cryptographic Failure in the web server, because of the support of obsolete, weak, or insecure ciphers in the configuration of TLS;
- A Cross-site WebSocket Hijacking flaw, caused by the application server code lacking verification of anti-CSRF tokens and only checking the session cookies before conceding access to resources through WebSocket;
- Excessive Privileges granted to the DB user of the application, as a result of a database server misconfiguration.

Besides the well-known range of enterprise web applications flaws, it must be considered that embedded

web applications may hide another range of vulnerabilities, i.e. memory corruption bugs. As a matter of fact, whereas enterprise application servers usually interpret server pages which are almost always written in memory-safe languages, embedded application servers are often made up of a single, or multiple, C/C++ compiled binaries.

This necessity stems from the resource constraints which embedded devices frequently need to satisfy, for instance in terms of power consumption or production costs. This implies that any query string parameter, HTTP header value, or body parameter, if not thoroughly validated by the application server, can lead to a memory corruption bug and, in the worst cases, result in the direct execution of code in the context of the application server process.

Consider, for instance, [CVE-2021-32941](#), caused by a memory corruption issue found on the Annke N48PBB NVR. As a matter of fact, the Annke N48PBB is exactly one of the aforementioned devices whose application server is implemented as a single, C/C++ compiled binary.

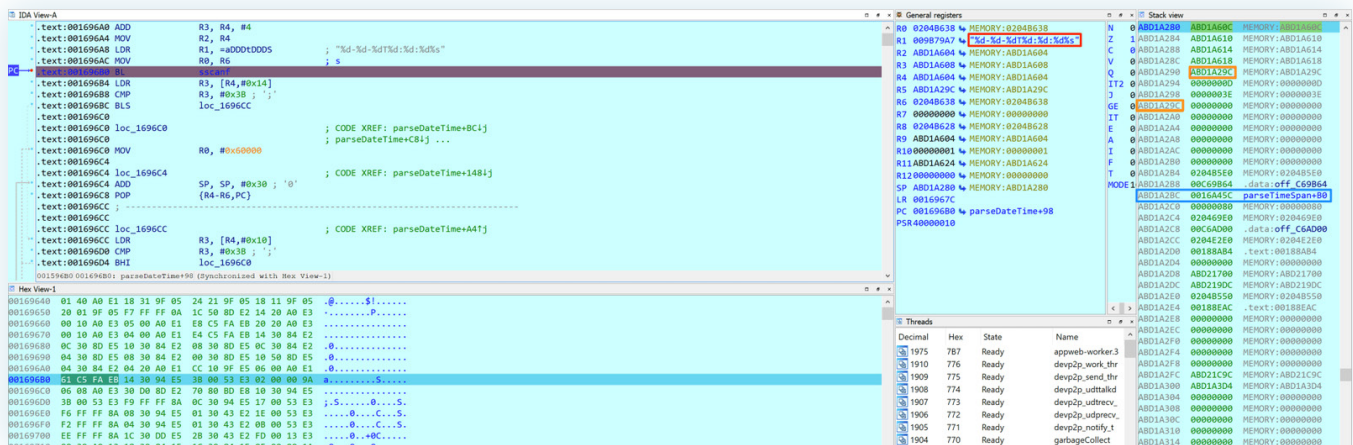


Figure 38 - CVE-2021-32941.

This vulnerability is due to a stack-based buffer overflow found in the playback search functionality. The functionality is accessible to all authenticated users by default, due to the usage of a `sscanf` function configured to write an

improperly validated HTTP body parameter into a limited-size buffer on the stack. In Figure 38, the format string is highlighted in red, the buffer address in orange, and the return address of the function in blue. Additionally, no

canaries are verified prior to performing the jump to the address, and a quick look at the output of “ps” confirms that the binary runs with root privileges on the device.

The final outcome of the memory corruption is a fully-fledged Remote Code Execution (RCE) with root privileges that, if exploited by a malicious operator or user, would result in their Elevation of Privilege (EoP) and full compromise of the system.¹³

When chained together, web application flaws can cause even more severe consequences to the security of a system. An example is given again by the just described Annke RCE. By itself, the stack-based buffer overflow causes an already significant impact to the security of the system, as low-privileged users may abuse it to obtain complete control of the device.

However, the same endpoint was also unprotected against CSRF attacks. By chaining these vulnerabilities together, a remarkably powerful attack primitive is obtained. An unauthenticated, external hacker is able to execute arbitrary code with root privileges on the device itself by convincing an administrator, operator, or user to browse a specifically crafted webpage while simultaneously logged in to the web interface of the device.

4.3 Services Supporting Remote Applications

The recordings stored in an NVR are generally made available in a few ways. The first is through the web management application, discussed in section 4.2.1. The second method involves a desktop or mobile application leveraging a dedicated service running on the NVR.

These services are sometimes implemented with proprietary protocols which at the very least provide a login process, in addition to serving content to the requesting application. This attack surface is exposed on the local network. Particular attention should be paid not only to

4.2.2 Remote Console

IP video surveillance devices sometimes offer remote console management interfaces, in addition to the previously discussed web-based option. This functionality is typically exposed through an SSH server, although sometimes even Telnet might be available.

There are two types of security issues that could affect these interfaces. The first concerns the implementation of the protocol itself. Embedded devices are seldom deployed with servers customized by the vendor to limit the actions of an authenticated users. Although vulnerabilities in relatively simple protocols such as telnet must not be factored out a priori, most of the focus should be devoted to the customizations.

A second and likely the most important security issue is credentials management. This issue emerges regularly from security assessments of remote console services. Hardcoded credentials are still found on a surprisingly regular basis, and sometimes remote access is configured to accept a key set by the vendor that cannot be managed by the end user. Before committing to a specific solution, asset owners should understand whether the devices under analysis have these insecure settings.

understanding the potential exposure of the credentials, but also to assessing the security of the audio/video streams in transit.

ONVIF is instead an open standard created to foster the interoperability between IP video surveillance products manufactured by different vendors. It defines a series of profiles addressing several features such as authentication and audio/video streaming, as well as (replace the comma with "and"), as well as device discovery and configuration.¹⁴ By virtue of being fairly broad, ONVIF touches several

attack surfaces, including management interfaces, remote applications, P2P, cloud video surveillance, and discovery services, which we discuss in this white paper.

An ONVIF compliant device will define the specific profiles that were implemented. Each feature will then behave according to the standard. For instance, streaming can be achieved through the RTSP protocol and vendors might chose to rely on known software components. Our suggestion when auditing an ONVIF target is to first identify the different pieces and then focus the attention on those that are custom to the vendor and not borrowed, for example, from open-source projects.

4.3.1 Dahua DVRIP

DVRIP is a proprietary protocol implemented by Dahua Technology in its devices to support desktop and mobile applications, such as SmartPSS. In 2020, a security researcher who goes by the moniker “Bashis” released a proof-of-concept capable of extracting in clear the credentials received from a DVRIP application.¹⁵ The vulnerability tracked with [CVE-2019-9682](#) is a perfect example of the issues commonly found in this type of protocol.

```
tools $ python3 Dahua-3DES-IMOU-PoC.py --poc_3des
[*] [Dahua 3DES/IMOU PoC 2020 bashis <mcw noemail eu>]
[+] Trying to bind to 0.0.0.0 on port 37777: Done
[+] Waiting for connections on 0.0.0.0:37777: Got connection from 192.168.1.4 on port 51908
[ ] Waiting for connections on 0.0.0.0:37777
[+] Client username: admin
[+] Client password: password
[*] Closed connection to 192.168.1.4 port 51908
[*] All done
```

Figure 39 - CVE-2019-9682.

4.4 P2P

Owners of a typical on-premise video surveillance solution seldom need to access the recording remotely through the internet. The usual solution to this requirement involves setting up a remote access solution, such as a VPN, and then accessing the NVR as if it were deployed on the local network relative to the client.

Peer-to-Peer (P2P), in the context of security cameras, refers to a functionality that allows a client to access audio/video streams transparently through the internet. This is achieved without configuring a firewall with a set of techniques broadly called “hole punching”. The presence of an internet reachable node called a P2P Server is crucial to this process. The P2P Server acts as a mediator, used by the client application and the NVR to establish bi-directional communication.

There are several proprietary implementations of this type of protocol, but in general the NVR has to act first and communicate its UID to the P2P Server, which univocally identifies a device within a P2P network, in addition to its IP/UDP port pair. The client will then be able to contact the P2P Server and ask the IP/UDP port pair of a given UID. Depending on a series of factors, the client might finally be able to authenticate directly with the NVR and access the recordings. Other times the whole communication is mediated by the P2P Server, which effectively acts as a man in the middle.

A P2P system exposes several internet reachable attack surfaces, mostly through the P2P Server.

From the point of view of an asset owner, a P2P Server represents a possible entry point within an internal

network.¹⁶ An attacker with knowledge of a UID can start hitting a device through the login process. UIDs were found to be enumerable in some implementations, with attackers consequently being able to start a bruteforce of the credentials.

Another potential vulnerability consists of having attackers impersonate an NVR with the goal of receiving the credentials from a client. In this scenario, malicious operators connect to the P2P Server advertising themselves as the proper NVR for a specific UID. A client looking for the matching UID will then provide the correct credentials to the attackers, who can later access the NVR successfully.

A further element to assess is the security of the audio/video streams as they traverse the internet. Nozomi Networks Labs has found vulnerabilities in the way two vendors implement this very feature.

4.4.1 Reolink P2P Vulnerabilities

Broadly speaking, a P2P protocol is composed of two logical parts. The first concerns the synchronization between a client, the P2P Server and an NVR, where for instance the client gathers the list of available recordings and then requests a specific audio/video stream. The second is about streaming the actual content of a recording.

In the Reolink implementation we found two separate vulnerabilities, one in each “section” of the protocol.¹⁷

Reolink uses an xml-based text protocol to exchange communications between actors in the P2P protocol. The vulnerability, tracked by [CVE-2020-25173](#), is about the usage of a single hardcoded key to encrypt the traffic. This means that an attacker with the knowledge of this key is effectively capable of recovering the cleartext of the communication. This is particularly worrisome because at the time of our analysis the complete credentials were transferred in clear, protected only by the hardcoded key.

The audio/video stream is instead sent in clear and by analyzing the UDP protocol it was possible to reconstruct the recording. This second security issue is identified by [CVE-2020-25169](#).

4.4.2 ThroughTek P2P Vulnerabilities

ThroughTek is a company that develops a P2P implementation, called “Kalay,” that is then shipped with several other vendors' products, such as Hikvision and Swann.

Earlier this year, we analyzed the network traffic generated by a Swann device during a P2P session. With the help of a debugger we found the library containing the protocol implementation and from there located the function containing the fixed key through which network packets are obfuscated.¹⁸

Although there is a binary rather than text-based protocol in the case of ThroughTek, the vulnerability is of the same type as in the Reolink implementation. [CVE-2021-32934](#) was assigned by CISA to track this vulnerability. While it's virtually impossible for an independent auditor to assess the actual number of devices affected, it must be noted that the vendor managing the platform, such as ThroughTek, has a complete understanding of the problem.

Mandiant recently disclosed a further security issue affecting the ThroughTek P2P protocol, identified with [CVE-2021-28372](#).¹⁹ This vulnerability allows an attacker to impersonate an NVR with the prerequisite of knowing a device UID, as we describe at the beginning of section 4.4.

4.4.3 P2P Deployment in Corporate Networks

As we have explained, P2P exposes a device deployed within a network to the internet, through a P2P server managed by an unproven third party. The biggest problem of this design lies not in allowing an attacker to access the recordings of a video surveillance system, but rather in the possibility of the target device being compromised by a remote malicious actor.

For this reason, P2P should not be deployed in corporate networks. Asset owners should take the necessary steps to verify that their video surveillance systems don't contain this feature.

4.5 Cloud Video Surveillance

Cloud video surveillance refers to all platforms where the role of a traditional NVR is taken by a SaaS platform. Since the cameras and the access controllers by design need to constantly reach the cloud provider through the internet, the networks where these devices are deployed tend to be designed with strong security, for instance leveraging a Zero Trust approach.

As devices are mostly managed through the SaaS platform, asset owners should focus on correctly managing the credentials and audit the access to the platform, in addition to carefully vetting the vendor. Removing the NVR from the equation doesn't mean that the devices being deployed shouldn't be tested for security vulnerabilities, as the cameras receive remote commands from the SaaS platform.

The most notable incident involving cloud video surveillance

was the breach that affected Verkada in March 2021.²⁰

Attackers gained initial access to an internet-exposed server used by the support team. From that system they managed to access both the recordings of customers as well as devices deployed on the field.

Deploying a cloud video surveillance system requires a robust network design, along with a network observability solution that is capable of profiling device behavior.

A big advantage of having a cloud platform to manage a fleet of devices concerns firmware updates. Updating firmware at scale is often a cumbersome process at which vendors seldom excel. With a SaaS solution, the firmware updates are available to be deployed as soon as they're released, dramatically reducing the time to patch.

4.6 Discovery Services

Discovery services are used by NVRs or device manager applications to automatically find, among the other assets, cameras and access controllers deployed within a network. These services come in two flavors – a vendor either designs a custom protocol or relies on a standard one such as WS-Discovery or UPnP, which are part of ONVIF.

Some products are shipped with both options. The custom protocol is typically preferred in a network populated only by devices and applications of a given vendor, while standard protocols are used in an heterogeneous environment.

4.6.1 Hikvision

Search Active Device Protocol (SADP) is an XML-based discovery protocol developed by Hikvision, a leading surveillance camera vendor, and shared by other OEMs. Since this protocol is implemented in desktop applications, security researchers can start their investigation simply by analyzing the network traffic and a Windows DLL.

Figure 40 shows the probe packet sent by the application to a multicast IPv4 address as seen on the network.

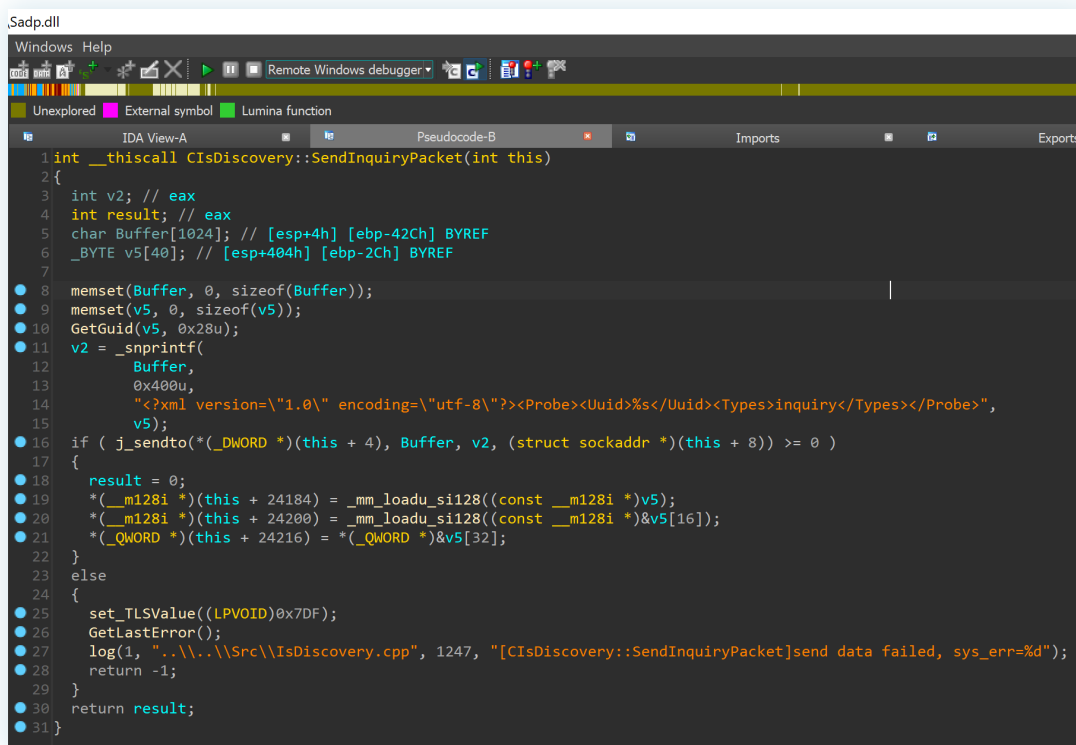
No.	Time	Source	Destination	Protocol	Length	Info	UDP Dst port
1	2021-04-22 11:08:27.200896	10.41.46.5	239.255.255.250	UDP	166	37020 → 37020 Len=124	37020 (37020)
Frame 2: 166 bytes on wire (1328 bits), 166 bytes captured (1328 bits) on interf: Ethernet II, Src: Winstars_04:79:16 (80:3f:5d:04:79:16), Dst: IPv4mcast_7f:ff:fa							
Internet Protocol Version 4, Src: 10.41.46.5 (10.41.46.5), Dst: 239.255.255.250							
User Datagram Protocol							
Source Port: 37020 (37020)							
Destination Port: 37020 (37020)							
Length: 132							
Checksum: 0x93ec [unverified]							
[Checksum Status: Unverified]							
[Stream index: 0]							
[Timestamps]							
UDP payload (124 bytes)							
Data (124 bytes)							

0000	01 00 5e 7f ff fa 80 3f 5d 04 79 16 08 00 45 00	..^....? l.y...E
0010	00 98 75 0d 00 00 01 11 00 00 0a 29 2e 05 ef ff	...u.....)....
0020	ff fa 90 9c 90 9c 00 84 93 ec 3c 3f 78 6d 6c 20<?xml
0030	76 65 72 73 69 6f 6e 3d 22 31 2e 30 22 20 65 6e	version="1.0" en
0040	63 6f 64 69 6e 67 3d 22 75 74 66 2d 38 22 3f 3e	coding=" utf-8"?>
0050	3c 50 72 6f 62 65 3e 3c 55 75 69 64 3e 33 30 39	<Probe>< Uuid>309
0060	39 32 41 36 36 2d 32 41 30 44 2d 34 30 39 41 2d	92A66-2A 0D-409A-
0070	41 41 36 44 2d 41 36 31 39 34 35 39 33 38 33 38	AA6D-A61 94593838
0080	35 3c 2f 55 75 69 64 3e 3c 54 79 70 65 73 3e 69	5</Uuid> <Types>i
0090	6e 71 75 69 72 79 3c 2f 54 79 70 65 73 3e 3c 2f	nquiry</ Types></
00a0	50 72 6f 62 65 3e	Probe

Figure 40 - Probe packet sent by SADP.

This instead is the function responsible for creating the probe, aptly named **SendInquiryPacket** (Figure 41).

Keep in mind that there's no guarantee that the implementation found in the firmware of a device corresponds to the one found in the desktop application.



```

Sadp.dll
Windows Help
Remote Windows debugger
Unexplored External symbol Lumina function
IDA View-A Pseudocode-B Imports Exports
1 int __thiscall CIsDiscovery::SendInquiryPacket(int this)
2 {
3   int v2; // eax
4   int result; // eax
5   char Buffer[1024]; // [esp+4h] [ebp-42Ch] BYREF
6   _BYTE v5[40]; // [esp+404h] [ebp-2Ch] BYREF
7
8   memset(Buffer, 0, sizeof(Buffer));
9   memset(v5, 0, sizeof(v5));
10  GetGuid(v5, 0x28u);
11  v2 = _snprintf(
12    Buffer,
13    0x400u,
14    "<?xml version='1.0' encoding='utf-8'><Probe><Uuid>%s</Uuid><Types>inquiry</Types></Probe>",
15    v5);
16  if ( j_sendto((_DWORD *)(this + 4), Buffer, v2, (struct sockaddr *)(this + 8)) >= 0 )
17  {
18    result = 0;
19    *(_m128i *)(this + 24184) = _mm_loadu_si128((const __m128i *)v5);
20    *(_m128i *)(this + 24200) = _mm_loadu_si128((const __m128i *)&v5[16]);
21    *(_QWORD *)(this + 24216) = *(_QWORD *)&v5[32];
22  }
23  else
24  {
25    set_TLSValue((LPVOID)0x7DF);
26    GetLastError();
27    log(1, "..\\..\\Src\\IsDiscovery.cpp", 1247, "[CIsDiscovery::SendInquiryPacket]send data failed, sys_err=%d");
28    return -1;
29  }
30  return result;
31 }
  
```

Figure 41 - SendInquiryPacket.

4.6.2 Axis

UPnP is among the set of discovery protocols supported by Axis devices. More specifically the implementation shipped in the firmware comes from the popular open-source project **libupnp**.²¹

From a security assessment perspective, this means that in addition to the usual binary analysis techniques that we've explored throughout this white paper, researchers can also leverage tools that require the source code such as AFL-style fuzzers.

american fuzzy lop 2.57b (harness_libupnp_xml)			
process timing		overall results	
run time : 0 days, 0 hrs, 0 min, 46 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 0 min, 0 sec		total paths : 636	
last uniq crash : 0 days, 0 hrs, 0 min, 0 sec		uniq crashes : 32	
last uniq hang : none seen yet		uniq hangs : 0	
cycle progress		map coverage	
now processing : 532 (83.65%)		map density : 2.42% / 4.80%	
paths timed out : 0 (0.00%)		count coverage : 4.20 bits/tuple	
stage progress		findings in depth	
now trying : splice 10		favored paths : 130 (20.44%)	
stage execs : 99/128 (77.34%)		new edges on : 179 (28.14%)	
total execs : 340k		total crashes : 383 (32 unique)	
exec speed : 1925/sec		total tmouts : 0 (0 unique)	
fuzzing strategy yields		path geometry	
bit flips : n/a, n/a, n/a		levels : 5	
byte flips : n/a, n/a, n/a		pending : 486	
arithmetics : n/a, n/a, n/a		pend fav : 30	
known ints : n/a, n/a, n/a		own finds : 610	
dictionary : n/a, n/a, n/a		imported : n/a	
havoc : 515/219k, 126/103k		stability : 97.93%	
trim : 29.96%/11.3k, n/a			

Figure 42 - AFL-style fuzzer output.

5. Conclusion

This white paper provides security analysts and researchers with a technical framework that helps assess the security posture of an IP video surveillance system and its vendor. This approach can be clearly abstracted and used with other complex systems that involve embedded devices and firmware.

One of the main theses presented is that access to unencrypted firmware images is paramount in the current landscape, where the danger of supply chain vulnerabilities is fully understood and potentially leveraged by malicious actors.

Though images are not always easily accessible, the hardware analysis techniques presented here can be used to extract the firmware from a device memory and to interact with its bootloader. We also demonstrate what it takes to analyze the actual executables after gaining access to the image.

Finally, we focused on decomposing common IP surveillance attack surfaces and touched upon specific vulnerabilities affecting different services. Given their prevalence and growing use of these systems, it's important to understand their security posture before deploying them on a network.

6. References and Further Reading

1. **"Size of the global video surveillance market between 2016 and 2025,"** Alsop, T., Statista, October 22, 2020.
2. **"A World With a Billion Cameras Watching You Is Just Around the Corner,"** Lin L., Purnell, N., The Wall Street Journal, December 6, 2019.
3. **"Ripple20 – 19 Zero-Day Vulnerabilities Amplified by the Supply Chain,"** JSOF, June 2020.
4. **"CVE-2021-3781,"** Ghostscript, September 9, 2021.
5. **"Supported Hardware,"** Flashrom, December 7, 2020.
6. **"Attify Badge,"** adi0x90, February 2, 2021.
7. **"Firmware Releases for All Our Supported Products,"** Axis Communications.
8. **"Dahua Product Security White Paper,"** Dahua Technology, 2020.
9. **"Hikvision OEM Directory,"** IPVM, June 3, 2021.
10. **"[MCR] Hikvision Packer/Unpacker for 5.3.x and Newer Firmware,"** Montecrypto, December 23, 2016.
11. **"Data Breach Investigations Report,"** Verizon, 2021.
12. **"Report: The State of Application Security in 2021,"** Campbell, A., Barracuda, May 18, 2021.
13. **"New Annke Vulnerability Shows Risks of IoT Security Camera Systems,"** Nozomi Networks Labs, August 26, 2021.
14. **"ONVIF™ Streaming Specification, Version 2.2,"** ONVIF, May 2012.
15. **"Dahua-3DES-IMOU-PoC.py,"** bashis, May 9, 2020.
16. **"Security Cameras Vulnerable to Hijacking,"** Marrapese, P., 2020.
17. **"New Reolink P2P Vulnerabilities Show IoT Security Camera Risks,"** Di Pinto, A., January 19, 2021.
18. **"New IoT Security Risk: Throughtek P2P Supply Chain Vulnerability,"** Nozomi Networks Labs, June 15, 2021.
19. **"Mandiant Discloses Critical Vulnerability Affecting Millions of IoT Devices,"** Valletta, J., Barzdukas, E., Franke, D., Mandiant, August 17, 2021.
20. **"Defending Against IoT Security Camera Hacks Like Verkada,"** Di Pinto, A., Nozomi Networks, March 12, 2021.
21. **"UPnP,"** Github.

Further Reading

- **"APT in a World of Rising Interdependence,"** Geer, D., NSA, March 26, 2014.
- **"BraveStarr – A Fedora 31 netkit telnetd remote exploit,"** Huizer, R., February 28, 2020.
- **"Mindshare: Dealing With Encrypted Router Firmware,"** Lee, V., February 6, 2021.
- **"This is Why People Fear the 'Internet of Things,'"** KrebsonSecurity, February 18, 2016.



Nozomi Networks

The Leading Solution for OT and IoT Security and Visibility

Nozomi Networks accelerates digital transformation by protecting the world's critical infrastructure, industrial and government organizations from cyber threats. Our solution delivers exceptional network and asset visibility, threat detection, and insights for OT and IoT environments. Customers rely on us to minimize risk and complexity while maximizing operational resilience.

© 2021 Nozomi Networks, Inc.

All Rights Reserved.

NN-WP-S3CUREC4M-8.5x11-001

nozominetworks.com