Proprietary Binary Protocols - An in-depth Analysis and Inspection of the DVRIP Protocol

A master's thesis presented

by

Thomas F. Vogt

to

The Department of Media and Information

in partial fulfilment of the requirements

for the degree of

Master of Science

in the subject of

Enterprise and IT-Security

University of Applied Science Offenburg

Offenburg

May 2020

ABSTRACT

Communication protocols enable information exchange between different information systems. If protocol descriptions for these systems are not available, they can be reverse-engineered for interoperability or security reasons. This master thesis describes the analysis of such a proprietary binary protocol, named the DVRIP or Dahua private protocol from Dahua Technology. The analysis contains the identification of the DVRIP protocol header format, security mechanisms and vulnerabilities inside the protocol implementation. With the revealing insights of the protocol, an increase of the overall security is achieved. This thesis builds the foundation for further targeted security analyses.

OFFICIAL DECLARATION

I hereby declare that I have not submitted this thesis in this or similar form to any other examination at the University of Applied Science Offenburg nor any other institution or university. I officially ensure that this paper has been written solely by me. I herewith officially ensure that I have not used any other sources but those stated by me. Any and every part of the text which constitutes quotes in original wording or in its essence have been explicitly referenced by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats. I also officially ensure that the printed version as submitted by me fully concurs with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination. Note this English translation, but only the official version in German is legally binding.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Hochschule Offenburg oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

_____	_____
Date	Thomas F. Vogt

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. Dr. rer. nat. Daniel Hammer and Prof. Dr. rer. nat. habil Dirk Westhoff for their consistent support and guidance during the running of this thesis. Special thanks also to the University of Applied Science Offenburg for providing parts of the analysed hardware, David Hulton from ToorCon Inc for providing one of the fastest available DES cracking systems worldwide and the Product Security Incident Response Team from Dahua Technology for the excellent communication in the vulnerability verification process.

Thank you to my girlfriend Anne-Kathrin, for all her love and support.

TABLE OF CONTENT

LIST OF FIGURES

1 INTRODUCTION

In the age of Industry 4.0 and the Internet of Things, the number of connected devices is increasing steadily. The security of these connected devices and systems is often better with known and open standards [1]. Unknown protocols and mechanisms remain less checked, which is why vulnerabilities are likely undetected or used by malicious actors [2]. Dahua Technology is one of the world's largest providers of surveillance systems [3]. They developed a proprietary binary protocol, named DVRIP or Dahua private protocol, for the management and control of artificial intelligence surveillance systems, network storage, smart building products, traffic signal control products or even drones [4], [5].

This protocol and the corresponding devices are accessible worldwide over the Internet. According to the IoT search engine *shodan.io* are more than 1.4 million entries directly accessible [6]. For malicious actors, this is a large attack surface. A vulnerability in the DVRIP protocol would have an enormous impact on these devices and their users. In the past, attackers used vulnerable IoT devices to build massive botnets and perform malicious actions with them [7]. This master thesis aims to analyse the DVRIP protocol to identify the protocol syntax and to examine the protocol for security vulnerabilities. Existing considerations to that protocol were only selectively researched and had insufficient depth most of the time. The results of the master thesis fill this gap.

Main Objectives

1. Identification of the DVRIP protocol structure
2. Identification of security mechanisms inside the protocol
3. Identification of security vulnerabilities inside the protocol design and implementation

After the initial explanation of the problem, the research question and the illustration of the scientific added value, the basics and theoretical background knowledge is illustrated in section 2. The section includes the general definition of network protocol reverse engineering, the structure of binary

protocols and the structure of text-based protocols. The typical causes of vulnerabilities are also shown.

Section 3 deals with the current state of research. There are explanations of already known methods and tools. This includes both the protocol reverse engineering and the current state of research regarding the DVRIP protocol. The research methodology is listed in section 4. In this section, a brief overview of the network setup is illustrated. Following that, the protocol identification methodology is explained in detail. The methodology for the reverse engineering process is also defined in this section. As a last primary process, the methodology for the analysis of the protocol implementation is defined.

Section 5 contains a detailed analysis of the protocol identification process. The section contains the creation and recording of protocol-relevant network traffic, the assignment of bytes to functions, the development of a Wireshark dissector and the development of a Dahua simulation client. After identifying the protocol syntax and developing a Dahua simulation client, this information is used to perform the security analysis. Section 6 contains the details about the reverse engineering process of the protocol. In this section, the general overview of the core protocol and operation software architecture is shown, and the primary login functionality is described. Also listed in this section are potentially interesting code parts of the Sonia[1] binary.

The protocol reverse engineering process is done during the protocol identification process and in combination with the implementation analysis of the protocol. Section 7 describes the protocol implementation analysis. For the analysis, a fuzzing and error detection client was developed to identify security-related vulnerabilities inside the DVRIP protocol implementation. The security analysis is done for the DVRIP protocol and also for parts of the RPC engine in the background.

[1] The Sonia binary application is the core application of the firmware

The research results are shown and described in section 8. This includes the identified DVRIP header format, the identified primary use-cases of the DVRIP protocol, the protocol design and corresponding security mechanism and the identified vulnerabilities. Section 9 summarises the master thesis research and shows the scientific added value and outlook of possible further research possibilities.

2 BASICS OF NETWORK PROTOCOLS

Network protocols are used to carry out a structured message exchange. In order to be able to communicate via protocol, this structural information must be available to the corresponding applications. If this information is not available, the protocol can be analysed to find out its structure. The necessary foundations for the analysis in Section 5, 6 and 7 are defined in the next subsections.

2.1 PROTOCOL REVERSE ENGINEERING TECHNIQUES

Network Protocols can be analysed in two main ways [8, p. 55]. The first approach is to look at the network level. Functions and actions are stored in protocol packets which are transmitted over the network. Intercepted packets can then be analysed, and byte fields are assigned to specific functions. The second approach is the possibility to understand the protocol on the application side. This is an attempt to find out which part of the program is used for which functions and how the network packets were generated. The focus within the master thesis lies on the analysis of network traffic.

2.2 CAPTURING APPLICATION TRAFFIC

Network packets can be recorded passively or using an active approach [9, pp. 11-35]. With the passive approach, the network traffic is recorded. With the active approach, an active intervention in the network communication is performed and packets can also be changed or manipulated. This is done either through a proxy or through a man-in-the-middle attack. The primary approach in this thesis was the passive recording of data, see subsection 5.1, which was realised with the open source applications `tcpdump` and `Wireshark`.

2.3 PROTOCOL STRUCTURES

Protocol structures define the basics of any protocol and follow a defined structure. A primary distinction is made between binary and text protocol structures. Forshaw gives a detailed overview [9, pp. 37-62] of that topic. Defined as the main elements are numeric data, booleans, variable-length data, bit flags or structured text formats. See Figure 1 as an excellent overview.

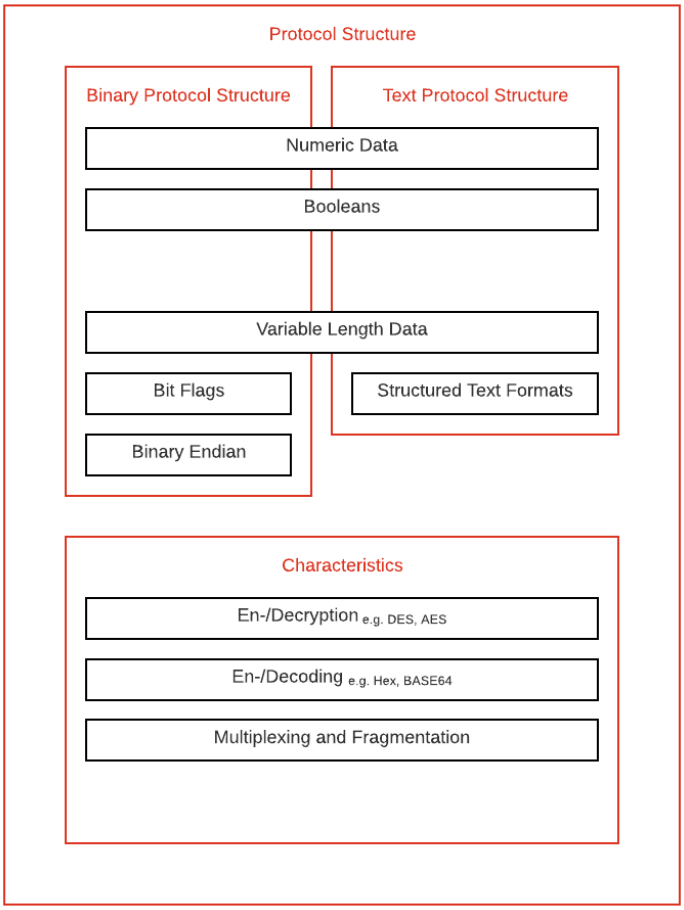


Figure 1: Protocol structure elements

With text protocols, the desired information can be displayed and also transmitted much more comfortably than with binary protocols. In binary protocols, unsigned integers, signed integers, variable-length integers or floating-point data are mainly used for numerical data. Boolean values

within binary protocols usually use 0 and 1, whereby a byte is often used for a Boolean value and not just a bit value. The transmitted data is usually data of variable length which can be represented, for example, by type, length, value combination. Bit flags are usually used for individual functions. In general, the end of the data can also be different. The interpretation of binary data can be either Little Endian or Big Endian. In principle, the data can also be transmitted encrypted or encoded. Multiplexing or fragmentation of data can also be considered in the general design of a network protocol.

2.4 NETWORK PROTOCOL SECURITY

The data transmitted via protocols contain information. It is highly likely that this information also contains sensitive data that must be protected. Secure protocols should meet the following criteria [9, p. 145]:

- Ensuring confidentiality so that it cannot be viewed
- Ensuring integrity so that it cannot be manipulated
- Protection against impersonification of server by implementing server authentication
- Protection against impersonification of client by implementing client authentication

The most commonly used mechanisms are [9, pp. 145-178] encryption algorithms like XOR, which is a crucial encryption. Random number generators are also ubiquitous for security-relevant processes. Symmetric Key Cryptography with, e.g. DES, 3DES or AES is often used to ensure security and confidentiality of the protocol and sensitive data. RSA is used as an Asymmetric Key Cryptography example. Signature algorithms, like MAC and HMACs, are used to ensure integrity checks. Public Key Infrastructure can ensure a higher security where e.g. X.509 certificates are used.

2.5 THE ROOT CAUSE OF VULNERABILITIES

According to the OWASP Internet of Things Top 10, 2018 [10], the following vulnerabilities are among the most common in IoT systems:

- I1 Weak or Hardcoded Passwords
- I2 Insecure Network Services
- I3 Insecure Ecosystem Interfaces
- I4 Lack of Secure Update Mechanism
- I5 Insecure or Outdated Components
- I6 Insufficient Privacy Protection
- I7 Insecure Data Transfer and Storage
- I8 Lack of Device Management
- I9 Insecure Default Settings
- I10 Lack of Physical Hardening

Our focus in this research is on I2: Insecure Network Service. Forshaw summarises the most common vulnerabilities associated with network protocols as follows [9, pp. 207-232]

- Remote Code Execution
- Information Disclosure
- Authentication Bypass
- Authorisation Bypass
- Memory Corruption Vulnerabilities
- Memory Exhaustion Attacks
- Storage Exhaustion Attacks
- Format String Vulnerabilities
- Command Injection
- SQL Injection

3 LITERATURE OVERVIEW

Duchêne et al. [8] describe in their work the tools and methods used from the last 12 years with a focus on inferring proprietary network protocols. The authors describe the necessary understanding and motivation behind network protocol reverse engineering. They describe motivators for protocol reverse engineering, which are: a) interoperability b) simulation of network protocols c) software security audit d) malware protocol analysis and e) network protocol conformance testing. In their work, they also talk about the benefits and disadvantages of some recent publications. The authors conclude that the following assumptions were made during the past development of reverse engineering tools: a) protocol reverse engineering based on network traffic was done various times in the past b) protocol reverse engineering tools based on network traffic focused on the analysis of vocabulary and grammar of the protocol c) new active approaches also focus on application inference d) since 2010 the research additionally focuses on binary reversing and debugging.

They describe the technical background and basic definition, which are used by the leading research publications on the topic of protocol reverse engineering. The researchers define the general definitions like protocol message types, protocol message fields, syntax and grammar types. With those defined components, information exchange is achieved. The authors mention the protocol reverse engineering challenges and the necessary steps for protocol reverse engineering, starting with identification and characterisation of the environment, observation steps, sanitising traces to obtain relevant messages and carrying out the inference for the protocol grammar.

In the detailed descriptions in their work, they briefly explain the chronological review of inference tools, both network and application side reversing. They also classify the identified tools in their work in different categories, like network- and application-based methods.

Significant in this research area is the work of Bossert, Guihéry and Hiet. In their publication *Towards automated protocol reverse engineering using semantic information* [11], [12]. Bossert describes his work as [11, p. 11]:

> *"[...] a practical approach for the automatic reverse engineering of undocumented communication protocols. This work leverages the semantic of the protocol to improve the quality, the speed and the stealthiness of the inference process when applied on complex protocols. Our work covers the two main aspects of the protocol RE, the inference of its syntactical definition (the protocol vocabulary) and of its grammatical definition (the protocol grammar). The algorithms we propose uses the semantic definition in both domains. We conducted multiple experiments to validate our approach by comparing previous state-of-the-art work against our algorithms. We also propose an open-source tool, called Netzob, that implements our work to help security experts in their protocol reverse engineering tasks. We claim Netzob is the most advanced published tool that tackles issues related to the reverse engineering and the simulation of undocumented protocols."*

The protocol reverse engineering, modelling and fuzzing framework Netzob offers the possibility to inference network traffic in an intuitive way. The protocol syntax is defined over so-called symbols, also shown in Figure 2. Each Symbol is a placeholder for a particular type of protocol message. An abstract Symbol can be specialised to bytes (sending) and bytes (received) can be abstracted into symbols. Figure 2 shows the conceptual approach of the mentioned methods.

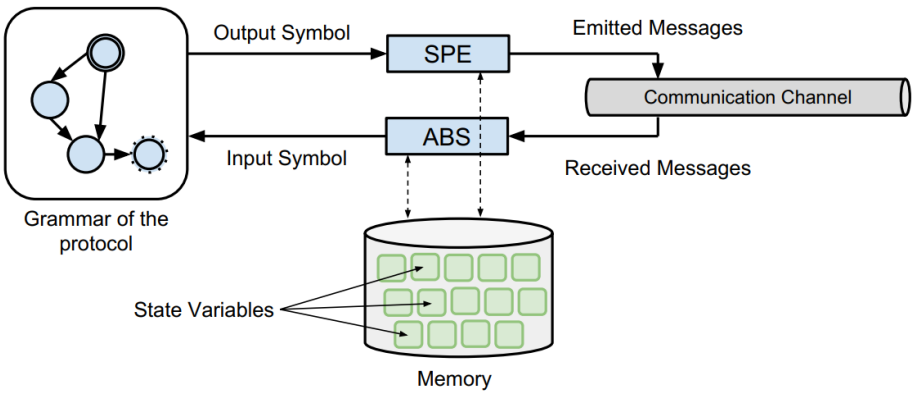


Figure 2: High-level Netzob overview [11, p. 83]

In the book „Attacking Network Protocols“ Forshaw [9] describes a hacker's guide to capture, analyse and exploit network protocols. He starts with the basic understanding of network protocols and their structures. Forshaw continues by explaining how network traffic can be captured and analysed and how applications can be reverse-engineered with these techniques. He also describes the typical security mechanisms of network protocols and the most existing types of vulnerabilities. First published in 2012, Forshaw also released a tool called CANAPE [13], which can be used to infer and support reverse engineering the protocols. Further information can be obtained in the whitepaper [14], on the slides [15] and video demonstration.

Different to the previously mentioned author, Bashis does not focus on the publication of new methods, approaches or tools to infer protocols. In the publicly accessible Github repository [16] the researcher has a collection of his security research on network cameras. This includes exploits and scripts for products of the vendors Axis, Dahua, Geovision, Realtek, Vivotek and others. In 2019 the researcher released a script [17] which provides an interface to the management console on Dahua products. This script operates with the DHIP protocol, where the structure of the DHIP protocol can be found. In the middle of January 2020, he was the first known researcher, who published a script [18] which uses the unknown DVRIP protocol. Some parts of the DVRIP structure can be retrieved from the script. The hash calculations used in the script for the DVRIP login process were used for the work of this thesis and marked accordingly. The hash calculations for the login process were used from that source, shown in subsection 5.5.

4 RESEARCH METHODOLOGY

The research in this thesis is mainly done with the defined methodology described in this section. As groundwork, a network environment is set, so that the analysis can optimally be done. The right network environment and network setup includes the correct usage of new and old target devices. With that set, the research follows a specific process of protocol identification and analysis, protocol reverse-engineering process and the analysis of the protocol implementation.

4.1 NETWORK ENVIRONMENT

A primary condition for the analysis is that it is done on a device with a new firmware. With such an approach, newly identified vulnerabilities are with a high probability undetected by Dahua Technology and are also not fixed in newer firmware versions. Therefore, the newest model of the device type IPC-HFW-1431S was ordered and shipped from China. For the newer firmware of products from Dahua Technology, no easy direct system access was known. For better accessibility and easier identification of vulnerabilities, an old device and firmware was used, which is IPC-HFW-1320S-W. The new model was shipped with the specification shown in Figure 3.

Device Type	System Version	WEB Version
IPC-HFW1431S	V2.800.0000004.0.R, Build Date: 2019-01-22	V3.2.1.688161
ONVIF Version	**S/N**	**Security Baseline Version**
16.12(V2.4.3.651299)	5C0726FPAGEC7C2	V1.4

Figure 3: Specification of IPC-HFW-1431S

For the reverse engineering process and the security analysis of the target protocol, multiple systems were used. The primary system was the Kali Linux operation system. On that all script development, analysis of network traffic and binary reversing was done. Also, a windows system was used to generate network traffic from windows-based configuration tools provided by Dahua. This includes the windows SDK demo applications. For the fuzzing process on a more extended period of time, a Raspberry Pi system was used, where scripts were executed and log data gathered. Figure 4 shows an overview of the network setup.

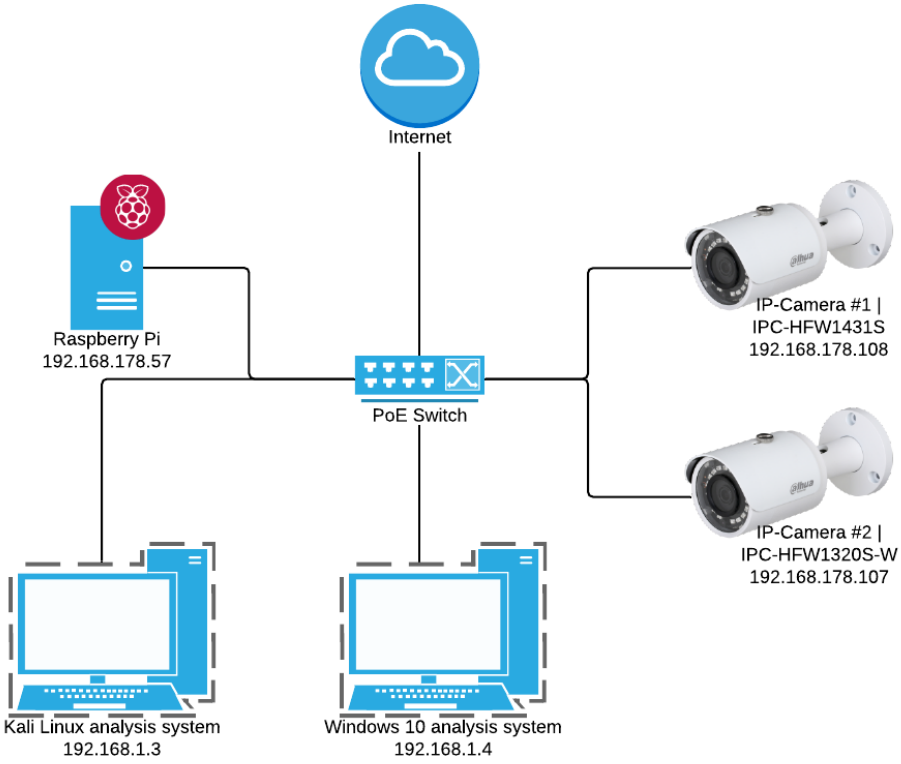


Figure 4: Network setup

4.2 PROTOCOL IDENTIFICATION AND ANALYSIS

Based on the current research and the own development of this thesis research, the protocol identification and protocol analysis were carried out after the following process scheme:

1) Implementing traffic capture setup
2) Development of a Wireshark dissector
3) Generation of application traffic
4) Bytes to function allocation
5) Implementing the protocol
6) Identification of the design
7) Protocol vulnerability research

For the first part of creating application traffic, the ConfigTool tool from Dahua was used, as well as the manufacturer's development SDK, with which various functions can be accessed. Wireshark was used to record network traffic. The recorded network packets were saved in a PCAP format for further analysis. The assignment of data packets or bytes to functions was primarily carried out using the Netzob framework. It allows the data packets to be displayed. In a next step, an attempt is made to simulate the network traffic recorded by specific functions. A specially developed client was used for this. The steps of this process were repeated.

4.3 REVERSE ENGINEERING

In the reverse engineering process, it was tried to identify protocol functions from the application side. The main application Sonia is responsible for the DVRIP communication. Note that the reverse engineering part is done in combination with the protocol identification and fuzzing of the protocol implementation.

In general, the analysis of the application was carried out using the following methodology:

1) Application overview analysis
2) DVRIP login analysis
3) High-value code analysis

As a first step, the general architecture of the target application was identified. With the gained knowledge, the primary functions in combination with the DVRIP login process were examined. This includes the research of how the bytes were processed inside the login functions. Also, a significant task was to identify functions and program code, the so-called "high-value" code, where possible relevant code is stored and processed.

4.4 ANALYSIS OF THE IMPLEMENTATION

The aim of analysing the implementation is to identify security-relevant vulnerabilities. This includes, for example, logic flaws or buffer overflow vulnerabilities. The analysis of the implementation was carried out according to the following process scheme:

1) Ensuring device monitoring for error detection
2) Custom fuzzing engine development
3) Smart fuzzing DVRIP endpoints
4) Smart fuzzing RPC over DVRIP
5) Smart fuzzing RPC over DHIP

The analysis of implementation documentation is structured according to the following scheme inside this thesis:

1) Description of the fuzzing endpoint
2) Description of the goal and possible results
3) Description of the implementation
4) Description of the observations and results

5 PROTOCOL IDENTIFICATION AND ANALYSIS

In this phase of the research, an attempt is made to identify and analyse the DVRIP network protocol. Subsection 5.1 describes how the applications network traffic was captured. In the next subsection 5.2, the developed Wireshark dissector is shown, which supports further analysis. In subsection 5.3, the details are shown, how and what application traffic was generated. This includes the development of a custom SDK C-code application. After that, subsection 5.4 shows how the allocation of bytes to functions were made and also what functions and actions could be determined to specific bytes. With the knowledge gained over the DVRIP protocol, subsection 5.5 describes the development and details of a protocol simulation script which was developed to simulate specific functions and actions on the target device. Following that step, subsection 8.1 contains the identified protocol design and security mechanisms. Based on that, subsection 8.1 contains the identified vulnerabilities on the DVRIP protocol.

5.1 CAPTURING NETWORK TRAFFIC

As described in subsection 2.2, the application can be captured actively and passively. In our research scenario, the traffic capturing process is focused on passive capturing. Based on the network environment, the best approach to capture traffic is on the target device itself. That was done for the camera system #2 with the older firmware, which allows direct system access via telnet on the target device. Capturing the traffic was then realised via `tcpdump` [19] and then copied to the Kali Linux analysis system for the later analysis. Directly capturing traffic on the device has the advantage that the network traffic can be captured but also the local loopback traffic can be captured. With this advantage, the DVRIP communication and architecture were identified, which is shown in subsection 6.1 in the overall application architecture of the primary protocol application `Sonia`. Based on the Windows- and Linux tailored configuration and management tools for products from Dahua technology, a Windows and Linux system were used to capture the generated traffic from the specific tools. This includes the execution and traffic capture from SDK demo examples, Dahuas

ConfigTool, and others. For that purpose, the widely-used network protocol analyser Wireshark was used [20].

5.2 DEVELOPMENT OF A WIRESHARK DISSECTOR

A Wireshark dissector [21] is a protocol definition for Wireshark to display a packet in a predefined structure which improves readability and leads to a better analysis within Wireshark. Due to the complexity of the DVRIP protocol, the developed Wireshark Dissector has a limited functionality and only shows the basic structure of the binary header. This is realised as a separation into 4-byte blocks of the transmitted data. The assignment of bytes to functions was primarily carried out and analysed via the Netzob framework, introduced in section 3 and shown in subsection 5.4. Figure 5 shows the DVRIP network communication inside the Wireshark tool, dissected with the custom DVRIP protocol dissector on the bottom side. Note that the protocol is shown as DAHUA in the figure.

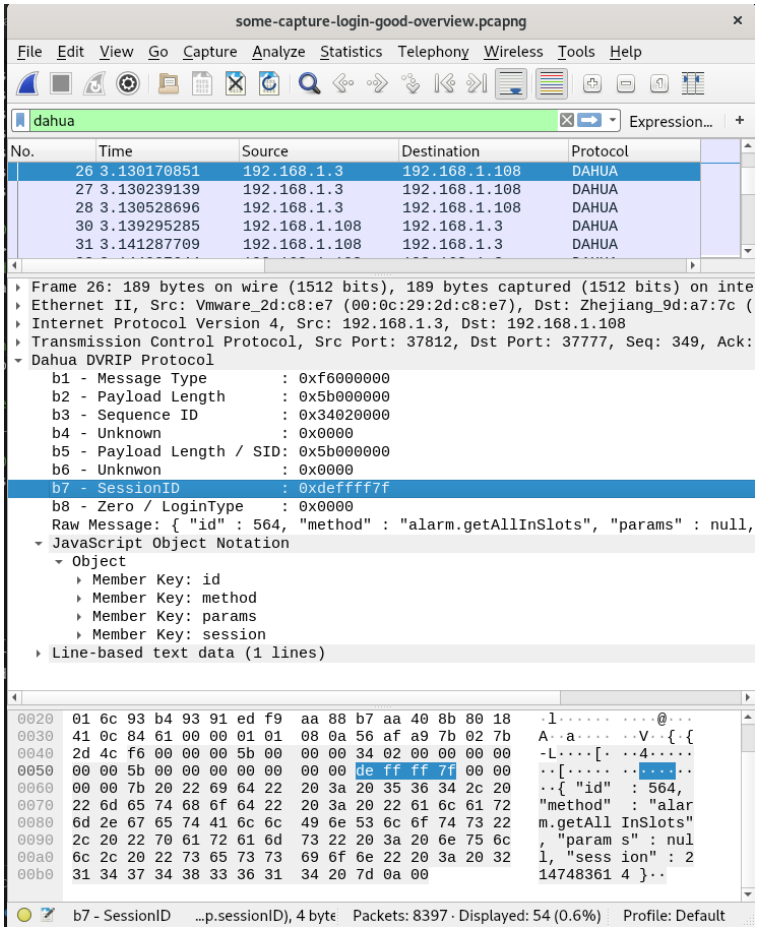


Figure 5: DVRIP Wireshark dissector

Figure 6 shows an excerpt of the Wireshark dissector written in the Lua programming language. First, the bytes were assigned to the corresponding tree items and then checked whether a JSON message was also included in the DVRIP network packet. Note that the naming convention of the byte blocks are based on the later identified DVRIP protocol header, shown in subsection 8.1. The full dissector code is available at [22] and also shown at appendix A3.

```
    -- naming the protocol in menu
    local subtree = tree:add(dahua_proto, buffer(), "Dahua DVRIP Protocol")

    -- adding the protocol fields
    local b1_tree = subtree:add(DVRIP_b1_message_type,   buffer(0,4))
    local b2_tree = subtree:add(DVRIP_b2_payload_length, buffer(4,4))
    local b3_tree = subtree:add(DVRIP_b3_sequence_id,    buffer(8,4))
    local b4_tree = subtree:add(DVRIP_b4_unknown_1,      buffer(12,4))
    local b5_tree = subtree:add(DVRIP_b5_payload_length_sid, buffer(16,4))
    local b6_tree = subtree:add(DVRIP_b6_unknown_2,      buffer(20,4))
    local b7_tree = subtree:add(DVRIP_b7_sessionID,      buffer(24,4))
    local b8_tree = subtree:add(DVRIP_b8_zero_type,      buffer(28,4))
```

```
    -- search for JSON in the payload field
    if buffer:len() >32 then
        -- detect and load JSON values
        payload_tvbrange = buffer.range

        if buffer(32,1):string() == "{" then
            -- loading buffer
            local tvb_uncompress = buffer(32,buffer:len()-32)

            -- raw text
            local b9_tmp = subtree:add(DVRIP_b9_payload_JSON_RAW, tvb_uncompress)

            -- as JSON structure
            local test = json:call(tvb_uncompress:tvb(), pinfo, subtree)

        end
```

Figure 6: DVRIP Wireshark LUA dissector excerpt

5.3 GENERATION OF APPLICATION NETWORK TRAFFIC

Application traffic was primarily generated using the Software Development Kit (SDK) and the ConfigTool. With the maintenance ConfigTool [23] Dahua offers the possibility to configure and control the camera, shown in Figure 7. This can be installed using the 'ToolBox' offered by Dahua. The SDK can also be downloaded from the Dahua homepage [24]. Necessarily, the record of network traffic and the execution of actions was carried out iteratively. Various actions were carried out via the ConfigTool, which had generated a massive amount of data. These were only used initially for the actual analysis. A more targeted option was the use of the SDK and a console script [18]. In the following, an overview of the SDK is given and then the essential relevant scenarios for generating application network traffic are described.

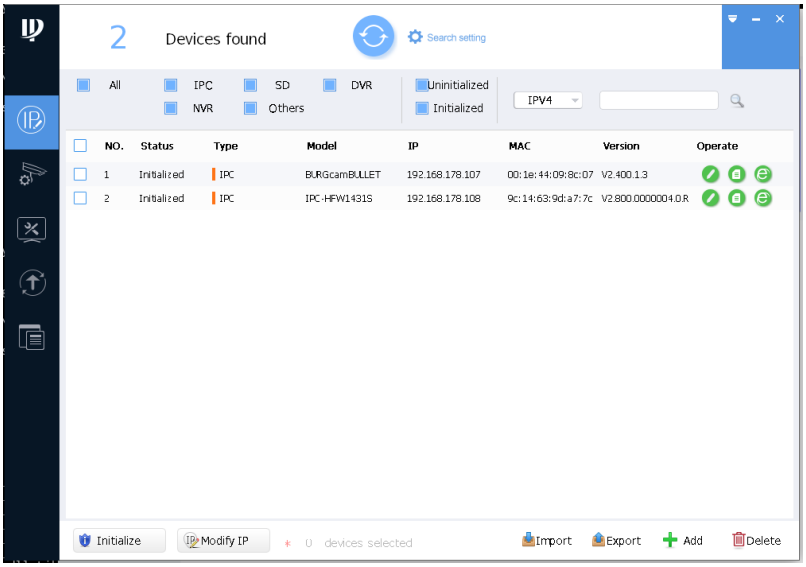


Figure 7: ConfigTool overview

5.3.1 OVERVIEW OF THE SDK

This chapter provides an overview of the Dahua SDK. It includes the existing documentation and data that are provided [24].

> *SDK is the software development kit used by software developer when developing DVR(digital video recorder), DVS(digital video server), IPC(IP camere) [sic], SD(speed dome), intelligent devices to monitor network application. The kit mainly includes the following functions: Device login, real-time monitoring, record playback and playback control, record download, PTZ control, audio intercom, video snapshpot [sic], alarm report, device search, intelligent event report and picture snapshot, user management and some other functions (device restart, device upgrade, video image parameter setup, channel name setup, device network parameter setup and etc.).*

The main elements of the Dahua SDK contain a PDF documentation with code examples for the general development. The code is provided as C-code. Also included are the C source header files (.h files) and the corresponding libraries which are required for the execution of the applications (e.g. .lib or .so files). The newest SDK was used for the analysis, which contains the following demo applications (excerpt, see Appendix A4 for all applications):

- AccessControl.exe
- DevInit.exe
- ImageTest.exe
- IntelligentDevice.exe
- IntelligentTraffic.exe
- NetSDKDemo.exe
- MonitorWall.exe

For the demo applications, the corresponding source code project was provided (Visual Studio project on Windows, QT Creator CentOS). The documentation contains interface descriptions for the individual functions. For example, the function `CLIENT_LoginEx2` is defined for the registration process and visualised in Figure 8. It should be noted that only the `CLIENT_LoginEx2` function can be called with parameters. It is unclear how the actual login process is implemented via the DVRIP protocol.

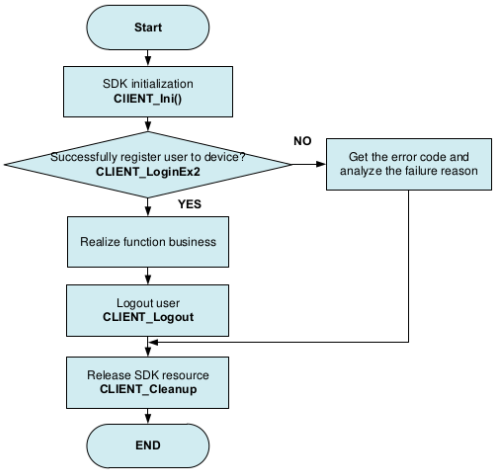


Figure 8: SDK login process overview

5.3.2 DEMO APPLICATION TRAFFIC

Almost every SDK demo application shown in the previous subsection and also listed in Appendix A4 was tested and used for the generation of application network traffic. Figure 9 shows the application for the `NetSDKDemo` binary application. The SDK demo application was executed both on the Windows Operating System as well as on the Linux Operating System. On the Linux system, the windows demo applications were also executed over `wine`.

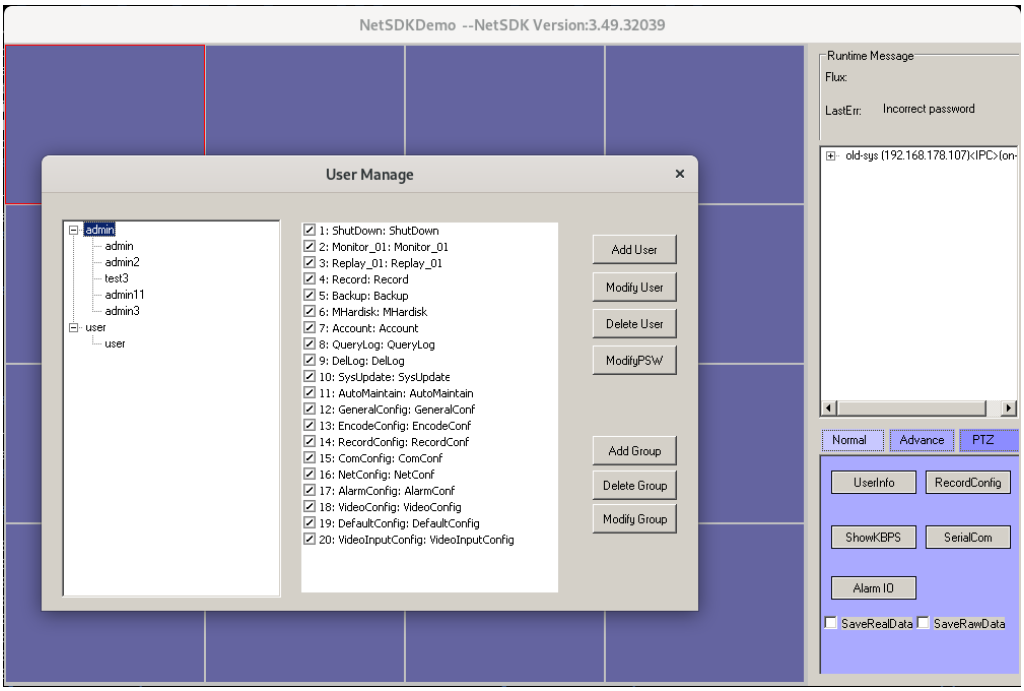


Figure 9: NetSDKDemo overview

5.3.3 CUSTOM APPLICATION TRAFFIC

To analyse the target DVRIP protocol and understand the information provided with the Dahua SDK, it is also important to develop custom SDK examples. Therefore, a basic login application was developed in C++. Figure 10 shows an excerpt of the C++ program with the main functionality of the login sequence. For a successful login in the target device, only a function with valid credentials is called. The actual processing of packets and calculations is done inside the SDK in closed-source

binaries. As the best compiling solution, the usage of a Makefile was identified, which is shown in Figure 11.

```
bool login(string destip, long port, string username, string password)
{
    cout << "login" << endl;
    int error = 0;
    NET_DEVICEINFO deviceInfo = {0};


    m_lLoginHandle = CLIENT_Login(destip.c_str(), port, username.c_str(), password.c_str(),
&deviceInfo, &error);

    if(m_lLoginHandle == 0)
    {
        if(error != 255)
        {
            cout << "login failed with code " << error << endl;
        }
        else
        {
            cout << "login also failed" << endl;
        }
        return false;
    }
    else
    {
        cout << "login ok" << endl;
        return true;
    }

}
```

Figure 10: Excerpt of a custom SDK login application

```
TESTCASE=testcase-001

CC=gcc
CXX=g++
RM=rm -f
CPPFLAGS=-g -Wall -std=c++11
LDFLAGS=-g -Wall -std=c++11
LDLIBS=-L/lib/dahua/ -ldhnetsdk
INCLUDES=-I.

SRCS=$(TESTCASE).cpp
OBJS=$(subst .cpp,.o,$(SRCS))

all: $(TESTCASE)

$(TESTCASE): $(OBJS)
        $(CXX) $(LDFLAGS) $(INCLUDES) -o $(TESTCASE) $(OBJS) $(LDLIBS)

$(TESTCASE).o: $(TESTCASE).cpp

clean:
        $(RM) $(OBJS)

distclean: clean
        $(RM) $(TESTCASE)
```

Figure 11: Makefile of custom SDK application

5.3.4 APPLICATION TRAFFIC FROM THE MANAGEMENT CONSOLE

As explained in section 3, the security researcher Bashis also identified parts of the DVRIP protocol during the time period of this thesis research and supplemented his existing script of the DHIP protocol with DVRIP parts [18]. The researcher's tool was also used to generate application traffic in order to examine it and to identify and reproduce the mechanisms. Figure 12 shows an example output of this tool.

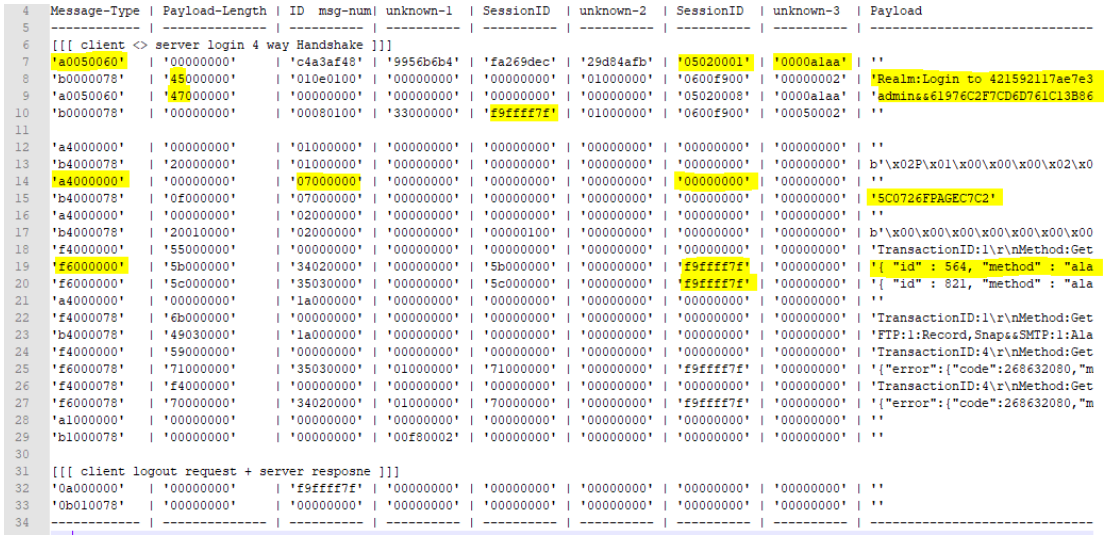
```
root@kali:/opt/r4bit/dahua/Tools
# python3 Dahua-JSON-Debug-Console-v2.py --rhost 192.168.178.108 --rport 37777 --auth admin:1234567a --proto dvrip
[*] [Dahua JSON Debug Console 2019,2020 bashis <mcw noemail eu>]
[+] Opening connection to 192.168.178.108 on port 37777: Done
[+] Dahua JSON Console: Success
[+] Login: Success
[+] Started keepAlive thread
[*] Remote device: IPC-HFW1431S
[Console]# user
17:41:06|[Manager] [ver:Unknown] info UserManager.cpp onConsole 2353 tid:373 User Name        Login As         Remote Address:
17:41:06|[Manager] [ver:Unknown] info UserManager.cpp onConsole 2354 tid:373 ____________________________________________________
17:41:06|[Manager] [ver:Unknown] info UserManager.cpp onConsole 2367 tid:373 admin            DVRIP            192.168.178.146
17:41:06|[Manager] [ver:Unknown] info UserManager.cpp onConsole 2367 tid:373 admin            Local            192.168.178.146
[Console]#
```

Figure 12: Script output of the device debug console

5.4 ALLOCATION OF BYTES TO FUNCTIONS WITH NETZOB

The recorded network traffic of the DVRIP configuration protocol runs on port 37777. The protocol reverse engineering, modelling and fuzzing framework Netzob [12], [25] was used to analyse this recorded network traffic. With that technique, a successful identification of the DVRIP protocol header was made. Also, executing a login sequence, logout sequence and executing binary and RPC calls were successful. Subsection 8.2 describes some of the identified use cases, which were implemented in a simulation script.

As described in section 3, Netzob [25] offers the possibility to analyse and edit recorded network traffic. The main benefit of this framework identified for this research is the representation of network traffic, as marked in Figure 13. This way, data packets can be analysed more clearly and easily. Abnormalities and attractive bytes can be identified more naturally, and these are coloured in yellow. The network traffic shown here reflects the communication process for the login and logout process via the DVRIP protocol (SDK). Each line represents the byte sequence either sent to or received from the target device.



```
Message-Type | Payload-Length | ID  msg-num| unknown-1  | SessionID  | unknown-2  | SessionID  | unknown-3  | Payload
------------ | -------------- | ---------- | ---------- | ---------- | ---------- | ---------- | ---------- | ------------------------------
[[[ client <> server login 4 way Handshake ]]]
'a0050060'   | '00000000'     | 'c4a3af48' | '9956b6b4' | 'fa269dec' | '29d84afb' | '05020001' | '0000a1aa' | ''
'b0000078'   | '45000000'     | '010e0100' | '00000000' | '00000000' | '01000000' | '0600f900' | '00000002' | 'Realm:Login to 4215921l7ae7e3
'a0050060'   | '47000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '05020008' | '0000a1aa' | 'admin&&61976C2F7CD6D761C13B86
'b0000078'   | '00000000'     | '00080100' | '33000000' | 'f9ffff7f' | '01000000' | '0600f900' | '00050002' | ''

'a4000000'   | '00000000'     | '01000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
'b4000078'   | '20000000'     | '01000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | b'\x02P\x01\x00\x00\x00\x02\x0
'a4000000'   | '00000000'     | '07000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
'b4000078'   | '0f000000'     | '07000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '5C0726FPAGEC7C2'
'a4000000'   | '00000000'     | '02000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
'b4000078'   | '20010000'     | '02000000' | '00000000' | '00000100' | '00000000' | '00000000' | '00000000' | b'\x00\x00\x00\x00\x00\x00\x00
'f4000000'   | '55000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | 'TransactionID:1\r\nMethod:Get
'f6000000'   | '5b000000'     | '34020000' | '00000000' | '5b000000' | '00000000' | 'f9ffff7f' | '00000000' | '{ "id" : 564, "method" : "ala
'f6000000'   | '5c000000'     | '35030000' | '00000000' | '5c000000' | '00000000' | 'f9ffff7f' | '00000000' | '{ "id" : 821, "method" : "ala
'a4000000'   | '00000000'     | '1a000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
'f4000078'   | '6b000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | 'TransactionID:1\r\nMethod:Get
'b4000078'   | '49030000'     | '1a000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | 'FTP:1:Record,Snap&&SMTP:1:Ala
'f4000000'   | '59000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | 'TransactionID:4\r\nMethod:Get
'f6000078'   | '71000000'     | '35030000' | '01000000' | '71000000' | '00000000' | 'f9ffff7f' | '00000000' | '{"error":{"code":268632080,"m
'f4000078'   | 'f4000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | 'TransactionID:4\r\nMethod:Get
'f6000078'   | '70000000'     | '34020000' | '01000000' | '70000000' | '00000000' | 'f9ffff7f' | '00000000' | '{"error":{"code":268632080,"m
'a1000000'   | '00000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
'b1000078'   | '00000000'     | '00000000' | '00f80002' | '00000000' | '00000000' | '00000000' | '00000000' | ''

[[[ client logout request + server resposne ]]]
'0a000000'   | '00000000'     | 'f9ffff7f' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
'0b010078'   | '00000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | '00000000' | ''
------------ | -------------- | ---------- | ---------- | ---------- | ---------- | ---------- | ---------- | ------------------------------
```

Figure 13: Inferred DVRIP traffic with Netzob

The various data packets have now been defined with the Netzob Framework, as illustrated in Figure 14. The functionality of the login process, the query for the serial number and session keepalive requests were defined in an inferring script with the usage of the Netzob framework. The whole inferring script is available on Github at [22].

```
CLIENT_LOGIN_INIT_SYMBOL = Symbol(name = "Client Login Initialization Symbol",
messages=messages_session1)
CLIENT_LOGIN_INIT_SYMBOL.fields = [
    Field(name="Message-Type",      domain=HexaString(b"a0050060")),
    Field(name="Payload-Length",    domain=HexaString(b"00000000")),
    Field(name="Command-ID",        domain=Raw(nbBytes=4)),
    Field(name="unknown-1",         domain=Raw(nbBytes=4)),
    Field(name="EXPLEN",            domain=Raw(nbBytes=4)),
    Field(name="unknown-2",         domain=Raw(nbBytes=4)),
    Field(name="SessionID",         domain=HexaString(b"05020001")),
    Field(name="unknown-3",         domain=HexaString(b"0000a1aa")),
    # Field(name="Payload",           domain=Raw(nbBytes=(0, 0))) Payload field is empty
]
INPUT_VOCABULARY.append(CLIENT_LOGIN_INIT_SYMBOL)
```

Figure 14: Netzob script excerpt, login symbol 1

As part of the research, it was discovered relatively quickly that the Netzob framework is rather poorly documented and limited in its functionality. Due to the insufficient documentation, it was tough to find the right functions for handling data packets, which was also very time-consuming. Through the intensive use of Netzob, some useful resources such as homepage backups, detailed Netzob documentation and example usages of Netzob libraries were identified, which were communicated to the community [26].

Due to the problematic usage of the Netzob framework, a separate Dahua simulation client was developed and used, see subsection 5.5. The identified bytes to function pairs are described as results in subsection 8.1.

5.5 DVRIP CLIENT DEVELOPMENT AND PROTOCOL ANALYSIS

In combination with the developed inference script, a new python client for the DVRIP protocol was developed. The aim of this script is to repeat use cases which were identified in the research before, described in subsection 8.1. This includes the successful login on a device, both with the 4-way and 2-way login handshake. Also executing direct binary commands and RPC commands should be possible.

Figure 15 shows the initial program flow determination from the developed python simulation client. Based on the specific parameters, the corresponding functions are executed. The simulation script also contains the fuzzing script engine.

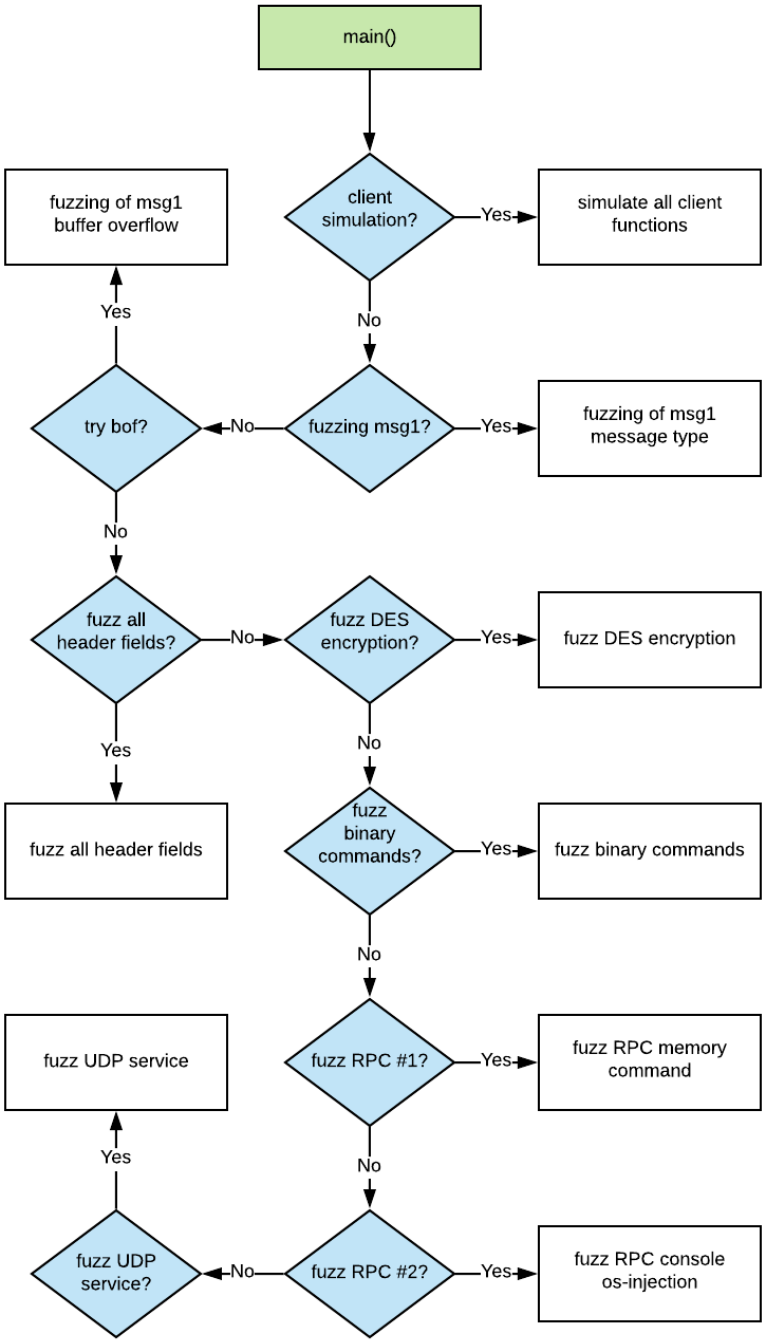


Figure 15: Protocol simulation and fuzzing script

As mentioned in section 3, the author Bashis also published parts of the DVRIP protocol in January 2020. In the 4-way handshake login process based on the detailed description in subsection 8.2.1, a hash generation script is used from the authors publication. A valid username and password are needed for a successful login. A successful login provides a session identifier, which can be used to issue binary and RPC commands on the target system. First, a DVRIP packet object for message one is created and sent to the target device. Message two is then received and also stored in a DVRIP packet object. With the knowledge of [18], described in section 3, the calculation of the Dahua hashes are made. See Figure 16 for details.

```
        logging.debug("\nCalculating Hash")
        myHash              =             calculate_hash_callback(DVRIP_PDU_msg2.b9_payload,
                 self.username, self.password)
        logging.debug(myHash)
        DVRIP_PDU_msg3  =  DVRIP_PDU(b'a0050060',  b'47000000',  b'00000000',  b'00000000',
b'00000000', b'00000000', b'05020008', b'0000a1aa', b'' )

        DVRIP_PDU_msg3.b9_payload = myHash
```

Figure 16: Hash generation function call

With the retrieved hash, message three is then constructed and sent to the target device. If the provided data is valid, the target device is returning a session identifier, which is stored in the local session.

Interesting Observations

During the execution and analysis with the simulation script, it was observed that when some basic RPC commands are issued, some specific system calls were made, which is shown in Figure 17. This indicates that the function matches the namespace, which can be used for later analysis. See also appendix A9 for a typical view of the log engine output.

```
13:50:57|[RemoteService] trace tid:3239 CSystemOperatorService::getDeviceType
13:50:57|[RemoteService]  trace  tid:3239  CSystemOperatorService::getDeviceType  successful
13:50:57|[RemoteService] trace tid:3243 CSystemOperatorService::getDeviceType
13:50:57|[RemoteService]  trace  tid:3243  CSystemOperatorService::getDeviceType  successful
13:50:57|[RemoteService] trace tid:3243 CSystemOperatorService::getSerialNo
13:50:57|[RemoteService] trace tid:3243 CSystemOperatorService::getSerialNo successful
13:50:57|[RemoteService] trace tid:3240 CSystemOperatorService::getVendor
13:50:57|[Manager] trace tid:3240 get Vendor: BurgWaechter
13:50:57|[RemoteService] trace tid:3240 CSystemOperatorService::getVendor successful
```

Figure 17: Functions called while executing RPC calls

6 REVERSE ENGINEERING DAHUA'S IOT SYSTEM APPLICATION

In order to better understand the DVRIP mechanisms and processes, in addition to the chosen network approach, there is the possibility to identify the protocol via the analysis of the firmware binary, which generates the traffic. Systems from Dahua Technology are primarily operated via a single main application, developed in C ++ or C code. This is the Sonia binary. There are already several weaknesses that can be assigned to the binary. See [27], [28] or [29] in general. The focus of the later analysis lies on relevant software elements that were examined during the course of the analysis.

Figure 18 illustrates the general architecture of the Sonia application. The figure was developed and updated frequently during the analysis of the DVRIP protocol. In general, the device offers three primary service endpoints to the external network. As the first significant observation, it was observed that the DVRIP protocol could be sent to different endpoints. This is over TCP port 80 and TCP port 37777. It was also identified in the reverse engineering process, that there should be a DVRIP service on TCP port 37776 for TLS encryption, but that could not be verified. Once a DVRIP message is sent to an open port, it is forwarded to the DVRIP Engine. The protocol engine then differs between direct binary commands and commands for the RPC server. Binary commands are issued directly in the Binary Command Engine. RPC commands are wrapped in the DHIP protocol and sent to DHIP Engine. The DHIP Engine then queries the RPC Server directly.

The Sonia binary represents Dahua's main system application, which was examined with the reverse engineering tool Ghidra [30]. An overview is shown in the next subsection 6.1.

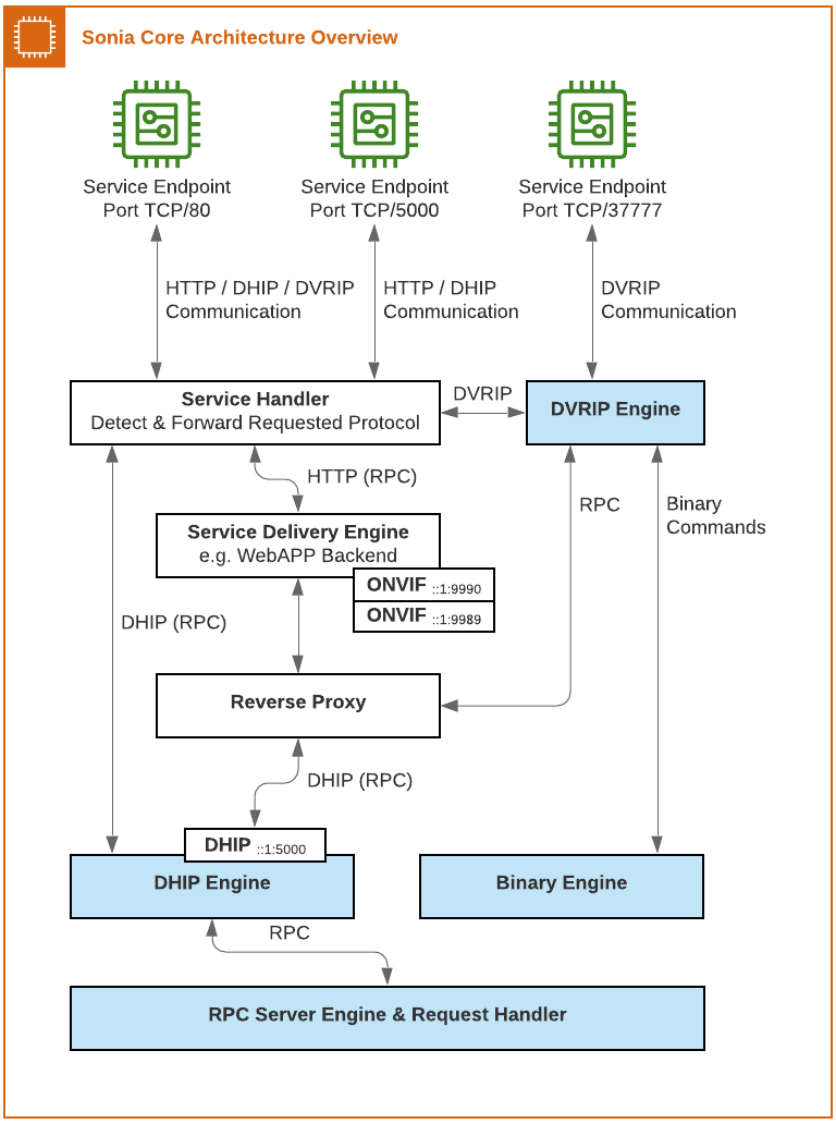


Figure 18: Sonia architecture overview

6.1 APPLICATION OVERVIEW

Due to the presence of debug information within the Sonia Binary mentioned in subsection 8.4.5, the original source code could very likely be restored. Class names of the C ++ source files, C source files, function and variable names are restored to the almost original code. The import summary from Ghidra shows the summary of the successful decompilation (excerpt[2]):

Project File Name:	sonia
Processor:	ARM
Endian:	Little
Address Size:	32
# of Bytes:	230519338
# of Defined Data:	211494
# of Functions:	70198
# of Symbols:	107864
# of Data Types:	462
# of ELF Source Files:	2881

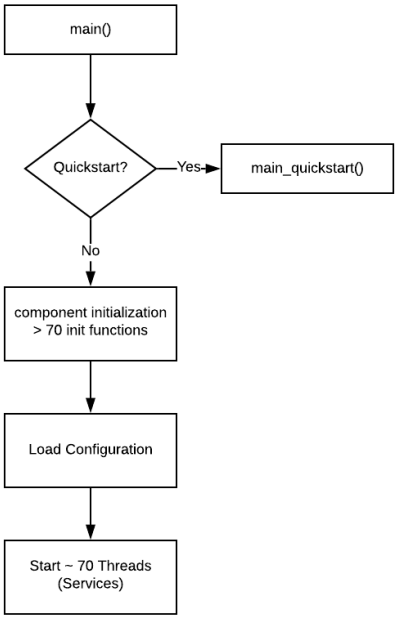


Figure 19: High-level Sonia application flow

It should be noted that the process is shown in a very simplified manner in Figure 19. As part of the analysis, only excerpts from the above functions and classes have been examined. The information provided in this section is intended to help classify the complexity and the following functions.

[2] The full import summary contains over 2000 lines and therefore it is not shown.

Another indication for the overall complexity is the so-called 'namespace' of the Sonia application. This includes around 300,000 entries from the Ghidra analysis. Microsoft's description for namespace is as follows:

> *A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organise code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier, for example std::vector<std::string> vec;, or else by a using Declaration for a single identifier (using std::string), or a using Directive for all the identifiers in the namespace (using namespace std;). Code in header files should always use the fully qualified namespace name. [31]*

Figure 20 shows a summary of the namespace `Dahua::DVRIP` and the number of entries at the start of each line. Every counted entry is for a unique C-function in a specific class.

```
146 CDVRIPConfigManager
 71 CmdsConfigUpdate
 58 CDHCommand0xF4
 41 CmdsDVRInfo
 30 CNetWorkService
 27 CPtzTransform
 22 CfgEncodeProcessor
 20 CNetDynamicManager
 18 CmdsAccount
 16 CSession
 16 CmdsF6Process
 16 CmdsCtrlProc
 15 CExtCfgInfoProcessor
 15 CDyOperation
 14 CVideoCaptureHelper
 12 CmdsUserLogin
 11 RecordProcess
 11 CmdsRecDownload
 10 CmdsDownloadFile
 10 CfgAgendaProcessor
```

Figure 20: Dahua::DVRIP namespace excerpt

In this phase of the reverse engineering process, the identification of the DVRIP login functionality and function or classes in high relation to the DVRIP were in focus. The whole application is due to the enormous number of classes and functions overly complex and not in focus.

6.2 DVRIP LOGIN FUNCTIONALITY

The DVRIP login functionality seems to be defined in the namespace `Dahua::DVRIP::CmdsUserLogin` and source file `Cmds_UserLogin.cpp`. Figure 21 shows the identified application flow once a DVRIP login message is retrieved on the target device.

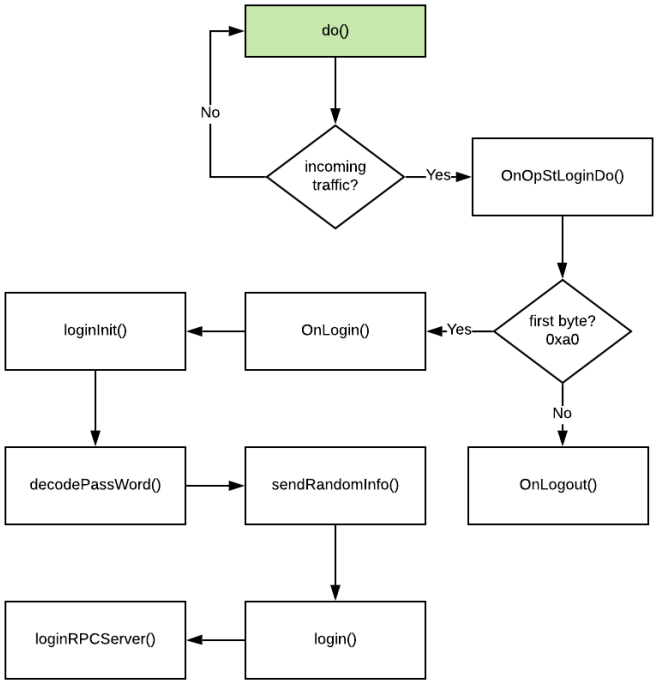


Figure 21: Cmds_UserLogin.cpp DVRIP application flow

6.3 INTERESTING FUNCTIONS AND CLASSES

During the reverse engineering process and security analysis, multiple functions and classes were investigated. In this subsection some of the functions or classes are described which had general interesting information or information which was used in the security analysis. Figure 22 shows the interesting Sonia code parts with a short description of what that code is most likely used for.

Class or function name	Short Description
NetWorkService.cpp	Handling for network service
Dahua::DVRIP::CNetWorkService::initDvrip()	Initialisation of DVRIP
Dahua::DVRIP::CNetWorkService:: setDefaultCfg()	Set default configuration for DVRIP
Dahua::DVRIP::CmdsUserLogin:: decodePassWord()	Used for password validation in DVRIP login process
Dahua::NetApp::IDeviceDiscoverServer:: handle_timeout	UDP discovery function
Dahua::DVRIP::Helper::DesEncrypt()	Encrypts stream of chars with DES
Dahua::DVRIP::Helper::SetKey	Set DES key
Dahua::DVRIP::CNetWorkService:: getCustomDVRIPCfg	Load DVRIP configuration data

Figure 22: Interesting Sonia code parts with a short descriptions

6.4 DHIP HEADER FORMAT

During the reverse engineering analysis, the DHIP header source structure file was identified, as shown in Figure 23. The information was identified in Ghidras Data Type Manager at sonia/DWARF/Cmds_F6Process.h/Dahua/DVRIP with the name DHIPHeader, source location at DWARF DIE: acod05 Cmds_F6Process.h:44. The naming of the byte blocks are based on the identified structure of that block.

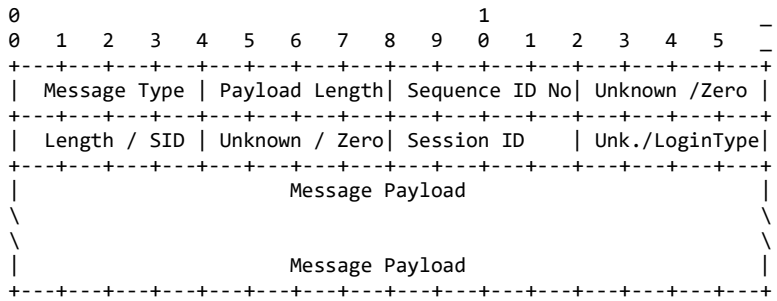


Figure 23: DHIP protocol header format

7 PROTOCOL IMPLEMENTATION ANALYSIS

During this phase of the research work, attempts were made to identify security-relevant weaknesses in the implementation of the DVRIP protocol. The functions implemented in section 5.5 are now examined for weaknesses and vulnerabilities. The focus of this analysis is on vulnerabilities, which can be exploited without valid credentials. The following endpoints were analysed:

1) DVRIP protocol engine
2) Fuzzing RPC over DVRIP
3) Fuzzing RPC over DHIP

7.1 DEVICE MONITORING AND ERROR DETECTION

The logging functionality form the Sonia interface was used to recognise and detect possible errors. All debug, error and other messages are displayed there. Figure 24 shows how the logging functionality of the Sonia application is called:

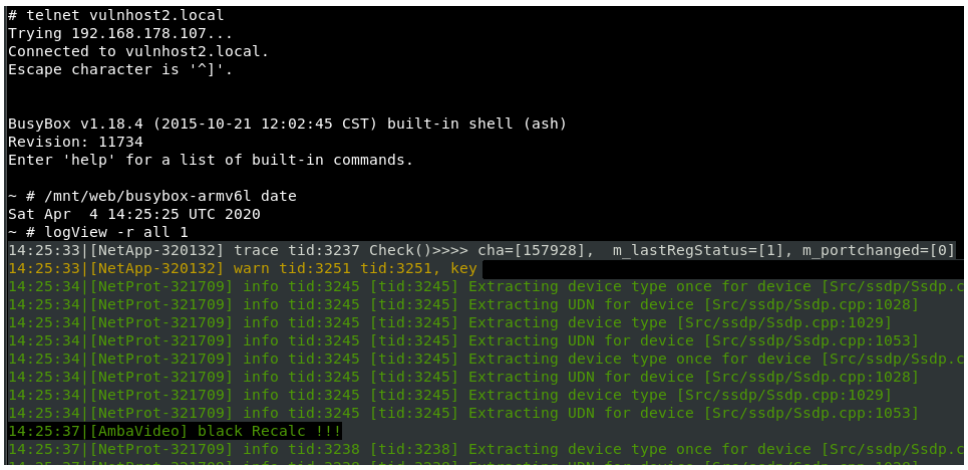


Figure 24: Logging output of the Sonia application

A logging engine was written for this purpose, see [22] for the full code. This establishes a connection with the target system and calls the logger. The output is fed back to the host and ultimately saved locally on the host. At the same time, the logging engine calls up this stored data and filters it

according to the standard output. Only relevant output is displayed, which allows optimal research on error message or other device behaviour. The strings for the standard filter rules are as follows:

- Has Been Deleted
- Connect RPCServer Successfully
- Extracting UND
- Extracting device type
- <P2PDevice.cpp:478>send heartbeat
- <P2PDevice.cpp:150>recv 200 OK
- key dhp2pP2P1.Burg.biz8800BurgBiz
- \[AmbaVideo\] black Recalc

7.2 CUSTOM FUZZING ENGINE

In order to be able to analyse the DVRIP protocol, the client simulation script was adapted, and a custom fuzzing engine was written. A full version of that fuzzing engine is available in this research GitHub repository [22]. The implemented use-cases are analysed with this fuzzing engine. Note that the identified DVRIP behaviour and byte sequences are based on the results from the protocol identification process shown in the result section 8. The defined fuzzing cases include as follows:

- Analysis of the DVRIP protocol
- Analysis of RPC commands over DVRIP
- Analysis of RPC commands over DHIP

7.3 FUZZING DVRIP

As described in subsection 8.2.1, the login process runs in a 4-way handshake. With the login sequence and the session identifier as the primary security mechanism, this is focused in the analysis. The next subsections will show an excerpt of the most investigated test cases. All executed fuzzing cases and endpoints can be accessed at [22].

The aim of the general fuzzing is described in subsection 2 and 4. Additionally focused is the general protocol behaviour when sending particular protocol messages or byte sequences, the detection of hidden functionality or error messages which indicate a unique type of vulnerabilities. Note that the identified DVRIP behaviour and byte sequences are based on the results from the protocol identification process shown in the result section 8.

7.3.1 CASE 1: ALL HEADER FIELDS (MSG1)

Fuzzing Endpoint

In this test case, msg1 from the 4-way handshake is used to analyse the target behaviour. According to the device state machine, the device should be in the state 'LOGIN-INIT' and listening for incoming messages.

Manipulation and expected results

In this fuzzing case all DVRIP header fields were used to try to trigger some errors on the target device. Figure 25 shows the manipulated header fields. Besides the overall expected results, it is expected to detect some unknown behaviour of the protocol.

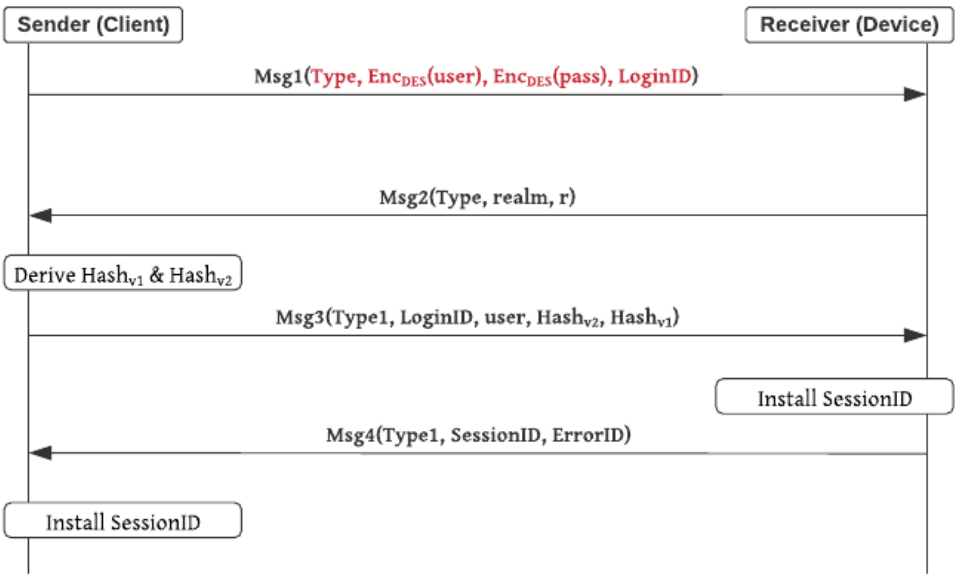


Figure 25: Sequence diagram with manipulated fields

Implementation details

According to Figure 25, the fields are manipulated and sent to the target device. In focus are the second 4 bytes of the DVRIP packet. Figure 26 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script.

```
        currentPDU.b1_msg_type            = binascii.hexlify(os.urandom(4))
        currentPDU.b2_pyaload_length      = b'00000000'

        currentPDU.b3_ID_msg_num          = binascii.hexlify(os.urandom(4)) #user
        currentPDU.b4_unknown_1           = binascii.hexlify(os.urandom(4)) #user
        currentPDU.b5_SessionID_RCV       = binascii.hexlify(os.urandom(4)) #pass
        currentPDU.b6_unknown_2           = binascii.hexlify(os.urandom(4)) #pass
        currentPDU.b7_SessionID_SEND      = binascii.hexlify(os.urandom(4))
        currentPDU.b8_unknown_login4way_ID    = binascii.hexlify(os.urandom(4))
```

Figure 26: Implementation details of fuzzing case 1

Observations and results

Observing and analysing the LogEngine output led to the assumption that some packets indicate a login try. This was combined with the known login sequence of the 4-way login sequence, which led to the 2-way login sequence. No successful exploitation of errors or relevant vulnerabilities were identified during this analysis. Figure 27 shows the output of the logging engine with different filters

in the windows. No successful vulnerability could be identified which could cause some relevant risks.

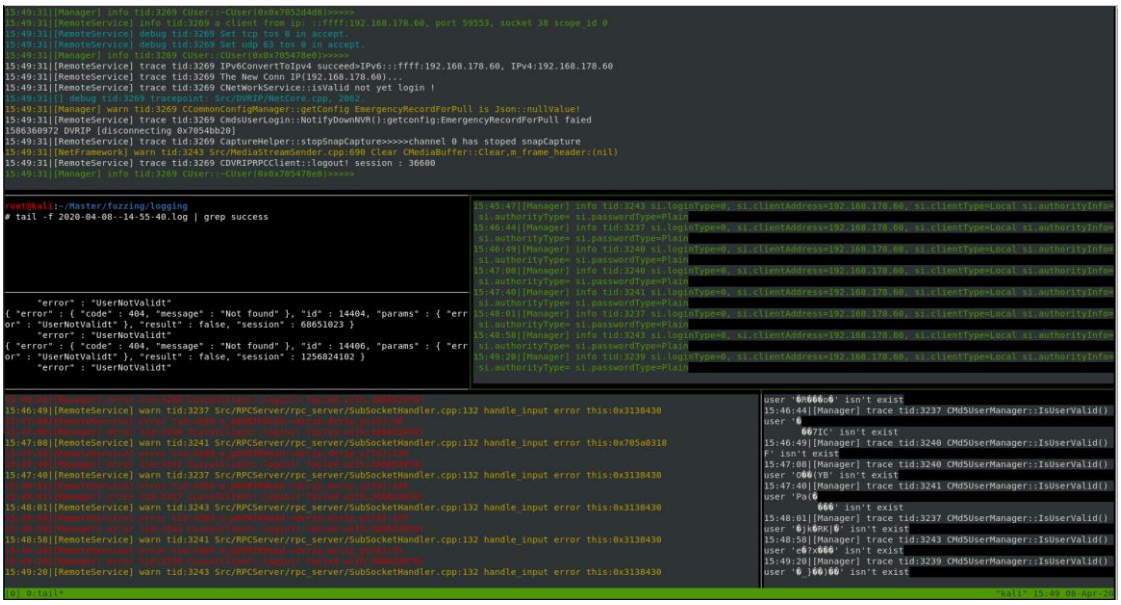


Figure 27: Fuzzing case 1 observation and logging view

7.3.2 CASE 2: MESSAGETYPE (MSG1)

Fuzzing Endpoint

In this test case, msg1 from the 4-way handshake is used to analyse the target behaviour. According to the device state machine, the device should be in the state 'LOGIN-INIT' and listening for incoming messages.

Manipulation and expected results

Figure 28 shows the manipulation of the message type. Besides the overall expected results, it is expected to detect some unknown behaviour of the protocol.

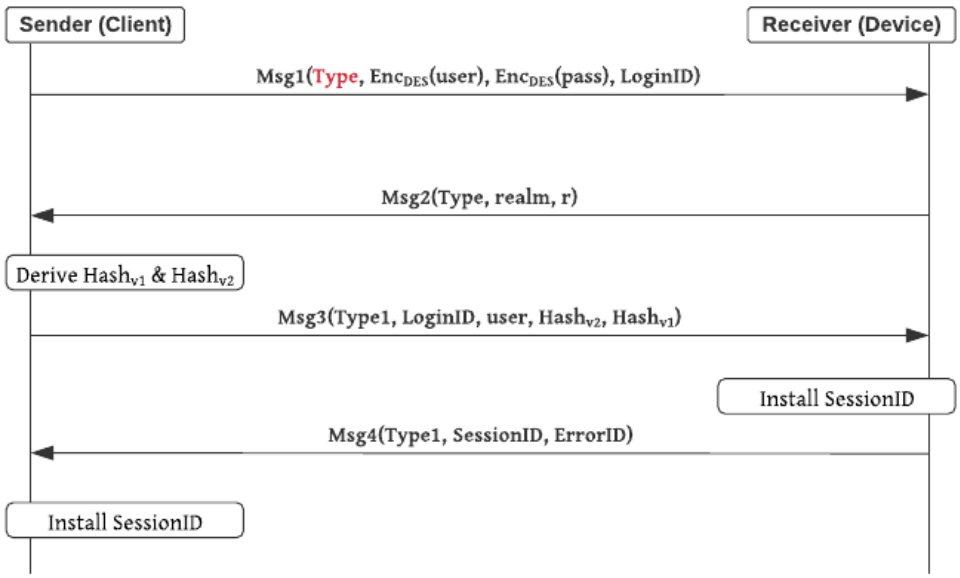


Figure 28: Sequence diagram with manipulated fields

Implementation details

According to Figure 28, the fields are manipulated and send to the target device. In focus are the second 4 bytes of the DVRIP packet. Figure 29 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script.

```
        b1_preFuzz                      = b'a0'
        fuzzy1                          = binascii.hexlify(os.urandom(3))
```

```
        currentPDU.b1_msg_type            = b''.join([b1_preFuzz, fuzzy1])
        currentPDU.b2_pyaload_length      = b'00000000'

        currentPDU.b3_ID_msg_num          = b'c4a3af48' #user
        currentPDU.b4_unknown_1           = b'9956b6b4' #user
        currentPDU.b5_SessionID_RCV       = b'6e6302f2' #pass
        currentPDU.b6_unknown_2           = b'b792f12c' #pass
        currentPDU.b7_SessionID_SEND      = binascii.hexlify(os.urandom(4))
        currentPDU.b8_unknown_login4way_ID    = binascii.hexlify(os.urandom(4))
```

Figure 29: Implementation details of fuzzing case 2

Observations and results

In some requests, it was detected that a session was created on the target device. Figure 30 shows an example of the monitoring process. In each different window shown, an individual log filter was set to detect different types of behaviour on the target device. In the windows with the red-marked x a few successful login sessions were identified during the analysis. That behaviour with further analysis led to the assumption that the target device allows a login with DES encryption. See subsection 8.2.2 for the 2-way DES login mechanism, which was developed for the simulation client as a feature. Traffic and username password combination were created with the custom SDK script, see subsection 5.3.3 for the usage of that script.

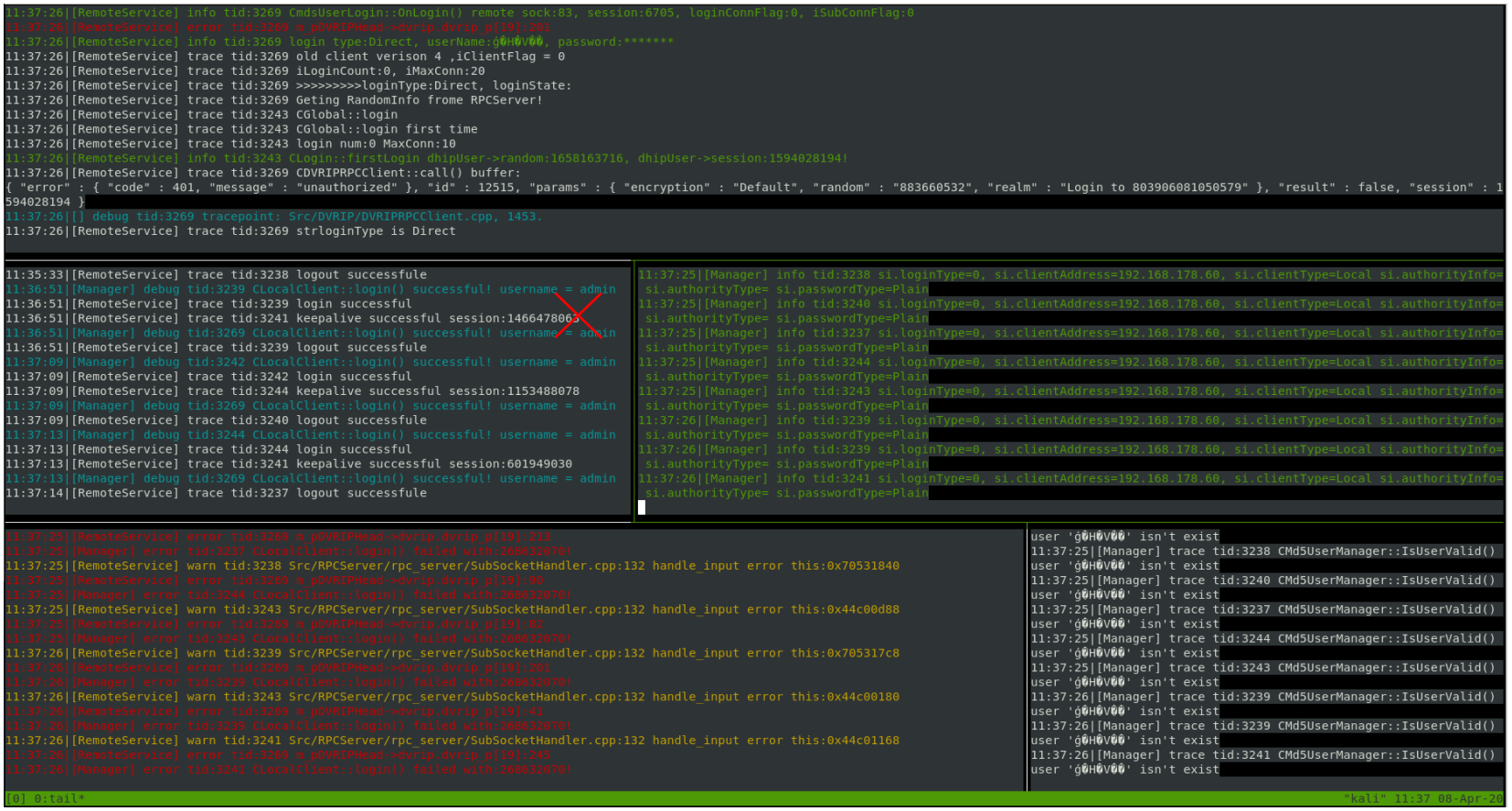


Figure 30: LogEngine output for fuzz case #2

7.3.3 CASE 3: BUFFER OVERFLOW INJECTION (MSG1)

Fuzzing Endpoint

In this test case, msg1 from the 4-way handshake is used to analyse the target behaviour. According to the device state machine, the device should be in the state 'LOGIN-INIT' and listening for incoming messages.

Manipulation and expected results

In this fuzzing case, the message length or payload length field is used to try to identify some buffer overflow vulnerabilities. The payload is also attached to the first sent message, shown as Payload in Figure 31.

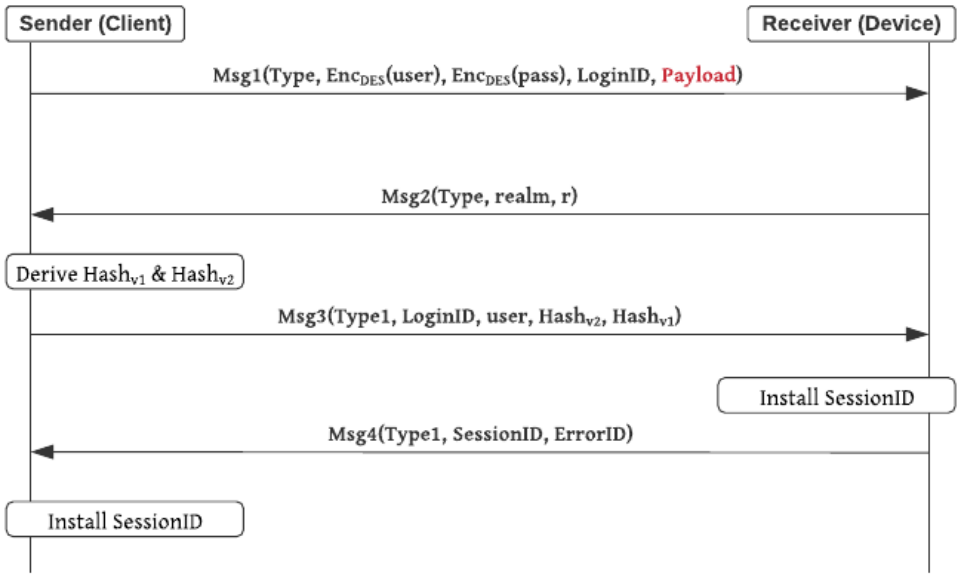


Figure 31: Sequence diagram with manipulated fields

Implementation details

According to Figure 31, the fields are manipulated and sent to the target device. In focus are the second 4 bytes and the attached payload in the DVRIP packet. Figure 32 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script.

```
        global fuzzLength
        payload = "A"*fuzzLength
        #print(payload)
        payloadLength = len(payload) #84 (int) => 54 in hex; hex(84):54
        payloadLength = struct.pack("<I", payloadLength)
        payloadLength = binascii.hexlify(payloadLength)

        # 0e6909badf8d9e4a aabbccdd

        # Complete randomized 4 bytes
        b1_preFuzz                          = b'a0'
        fuzzy1                              = binascii.hexlify(os.urandom(3))
        currentPDU.b1_msg_type              = b''.join([b1_preFuzz, fuzzy1])

        #self.b1_msg_type                   = binascii.hexlify(os.urandom(4))
        currentPDU.b2_pyaload_length        = payloadLength #b'04000000'
        #self.b2_pyaload_length             = b'00000000'

        currentPDU.b3_ID_msg_num            = binascii.hexlify(os.urandom(4)) #user
        currentPDU.b4_unknown_1             = binascii.hexlify(os.urandom(4)) #user
        currentPDU.b5_SessionID_RCV         = binascii.hexlify(os.urandom(4)) #pass
        currentPDU.b6_unknown_2             = binascii.hexlify(os.urandom(4)) #pass
        currentPDU.b7_SessionID_SEND        = binascii.hexlify(os.urandom(4))
        currentPDU.b8_unknown_login4way_ID  = binascii.hexlify(os.urandom(4))

        # payload length must be written in b2
        currentPDU.b9_payload               = payload

        logging.debug(currentPDU.state())

        fuzzLength = fuzzLength * 2
        print(fuzzLength)
```

Figure 32: Implementation details of fuzzing case 3

Observations and results

During the analysis and in the investigation of the network traffic and also the logfiles, it was observed that there is some "valid packed" detection on the target side. That means that when the DVRIP packet has an invalid length, no response is sent back to the client.

Feature detected: Invalid Packet detection.

Triggering a buffer overflow was not successful, no suspicious logging content was observed, nor crash was detected. The buffer overflow was tested with a max buffer size of 5242880 chars. Figure 33 shows the output of the logging engine.

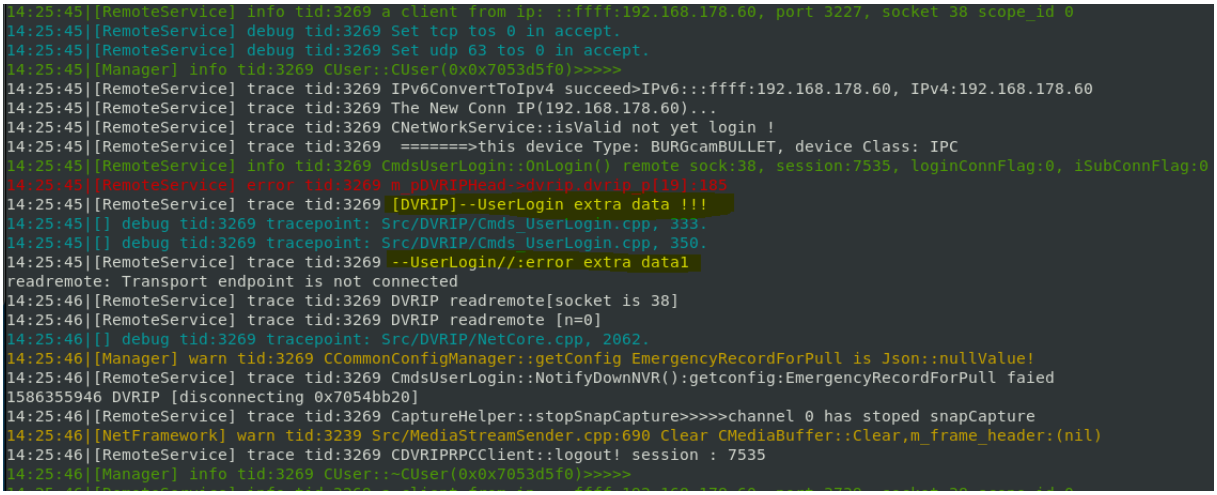


Figure 33: Logging engine output for fuzzing case 3

7.3.4 CASE 4: SEARCHING DESKEY MSG1

Fuzzing endpoint

As discovered in the previous test cases, it was observed that sometimes the DES login handshake was successful. In this fuzzing case, the decryption engine on the target device is the fuzzing endpoint. The results from the decryption engine can be accessed via the logging engine. Packets can be directly sent via a manipulated msg1 from the 4-way handshake. According to the device state machine, the device should be in the state 'LOGIN-INIT' and listening for incoming messages.

Manipulation and expected results

In particular, the username fields as header fields 3 and 4 and the password in header fields 5 and 6 were manipulated, shown in Figure 34. A successful encrypted sent message would be identified with the cleartext username 'admin' in the LogEngine and thus a piece of evidence that the right DES key was identified. If none of the keys provided from crack.sh is working, there is a high probability that the DES implementation in the Sonia binary is broken.

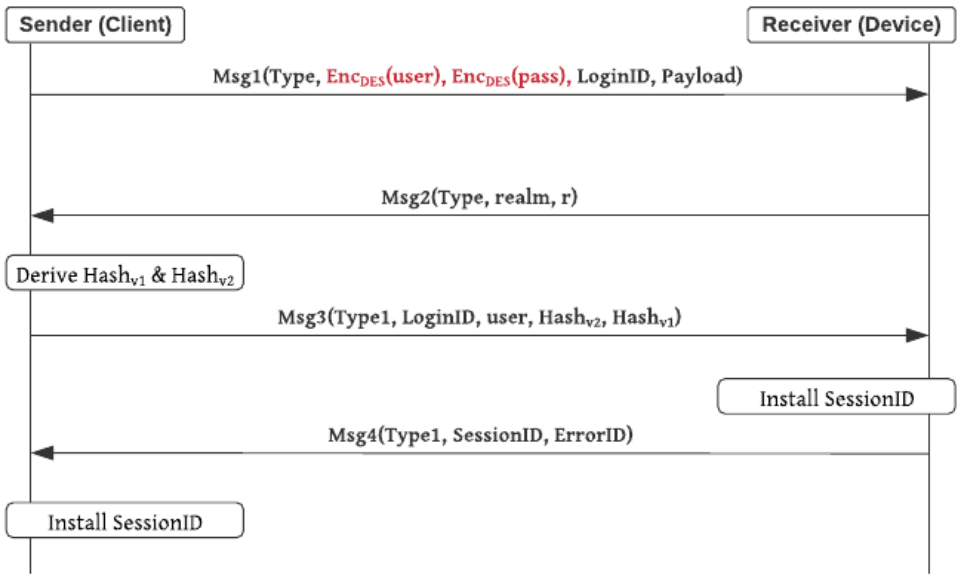


Figure 34: Sequence diagram with manipulated fields

Implementation details

To crack the DES key, the following data was provided for the cracking process via crack.sh:

```
# Pair #1
# Note: Cipherstream: 6e0f14f6dfa85cda
# Note: Cleartext: aaaaaaaa

# Submit an Encrypt Job:
# ./des_kpt.py parse -p 6161616161616161 -m ffffffffffff0000 -c 6e0f14f6dfa85cda -e
                 PT = 6161616161616161
                  M = ffffffffffff0000
                 CT = 6e0f14f6dfa80000
                  E = 1
crack.sh Submission = $97$YWFhYWFhYWH///////8AAG4PFPbfqAAA
```

Figure 35: Submission job for DES brute force, pair #1

```
# Pair #2
# Note: Cipherstream: fa269dec29d84afb
# Note: Cleartext: 1234567a

# Submit an Encrypt Job:
# ./des_kpt.py parse -p 3132333435363761 -m ffffffffffff0000 -c fa269dec29d84afb -e
                 PT = 3132333435363761
                  M = ffffffffffff0000
                 CT = fa269dec29d80000
                  E = 1
crack.sh Submission = $97$MTIzNDU2N2H///////8AAPomnewp2AAA
```

Figure 36: Submission job for DES brute force, pair #2

Figure 35 and Figure 36 is showing the generation of the test tokens for the cracking process. The scripts can be accessed via [32]. The results retrieved via the cracking service are shown in Figure 37 and Figure 38. The keys provided are for each cracking job around 200 entries, because only 56bit of the 64 bit were used in the DES encryption and also some possible collisions were collected.

```
Crack.sh has successfully completed its attack against your known
plaintext encrypt parameters. A list of the valid keys are attached
and can be verified using the 'des_kpt' tool:

Token: $97$MTIzNDU2N2H///////8AAPomnewp2AAA

$ ./des_kpt.py encrypt -p 3132333435363761 -k <key>
...
                 CT = fa269dec29d80000
...

This run took 113186 seconds. Thank you for using crack.sh, this
concludes your job.
```

Figure 37: Result notification for known plaintext DES brute force

```
Crack.sh has successfully completed its attack against your known
plaintext encrypt parameters. A list of the valid keys are attached
and can be verified using the 'des_kpt' tool:

Token: $97$YWFhYWFhYWH///////8AAG4PFPbfqAAA

$ ./des_kpt.py encrypt -p 6161616161616161 -k <key>
...
                CT = 6e0f14f6dfa80000
...

This run took 113166 seconds. Thank you for using crack.sh, this
concludes your job.
```

Figure 38: Result notification for known plaintext DES brute force

With the around 400 provided keys ciphertexts were generated for the cleartext 'admin'. The encryption on the device should, with the right key, return 'admin' in the LogEngine. The generated DVRIP packets were defined according to Figure 39. See [22] for the full Fuzzing Engine Script.

```
        global deskey

        # Complete randomized 3 bytes
        b1_preFuzz                          = b'a0'
        fuzzy1                              = binascii.hexlify(os.urandom(3))
        currentPDU.b1_msg_type              = b''.join([b1_preFuzz, fuzzy1])

        currentPDU.b2_pyaload_length        = b'00000000'

        currentPDU.b3_ID_msg_num            = deskey.encode()[:8] #user
        currentPDU.b4_unknown_1             = deskey.encode()[8:16] #user
        currentPDU.b5_SessionID_RCV         = binascii.hexlify(os.urandom(4)) #pass
        currentPDU.b6_unknown_2             = binascii.hexlify(os.urandom(4)) #pass
        currentPDU.b7_SessionID_SEND        = binascii.hexlify(os.urandom(4))
        currentPDU.b8_unknown_login4way_ID  = binascii.hexlify(os.urandom(4))
```

Figure 39: Implementation details of fuzzing case 4

Observations and results

As the main result, it was observed that none of the keys seem to be working because all decrypted usernames were most of the time no ASCII text or not the correct cleartext 'admin'. See Figure 40 for an overview. With the right DES key and thus generated ciphertext, the username 'admin' would be visible in that view. This assumption was proved with the release and publishing of the broken DES implementation vulnerability from Bashis [33] in early May 2020.

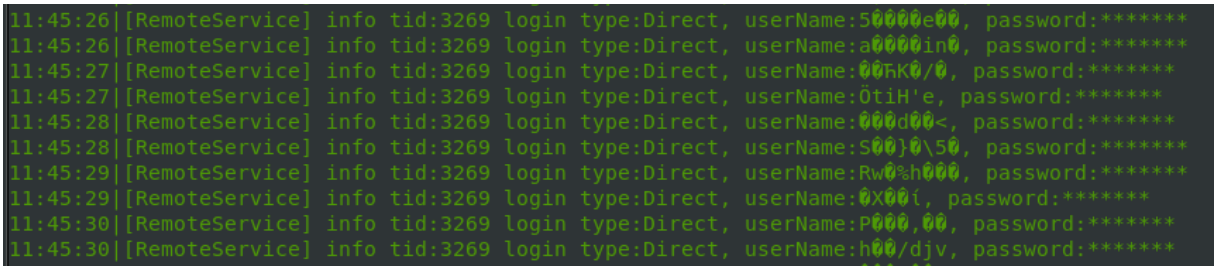


Figure 40: Log engine view of the DES key search

Also, it was observed that different login types were triggered on the device side. This includes the login types DES_LOGIN, DIRECT, LDAP and Active Directory. The login type DIRECT seems to be the standard login type used from the SDK.

7.3.5 CASE 5: BINARY ENGINE BEHAVIOUR, PDU FUZZ

Fuzzing endpoint

In this test case, the BinaryEngine on the target device is the fuzzing endpoint. A valid session should be retrieved over the 2- or 4-way login handshake.

Manipulation and expected results

With the manipulation of the CommandId DVRIP header field, it was tried to manipulate specific commands and to identify new unknown commands. See Figure 41 as an overview.

Figure 41: Sequence diagram with manipulated fields

Implementation details

According to Figure 41, the fields are manipulated and sent to the target device. In focus are the third 4 bytes of the DVRIP packet which specifies the command to execute with a numeric number or hex value. Figure 42 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script.

```
        code = struct.pack("<I", code)
        code = binascii.hexlify(code)

        currentPDU.b3_ID_msg_num          = code
```

Figure 42: Implementation details of fuzzing case 5

Observations and results

The first byte is accessible for some commands, see Figure 43 for examples. With that command iteration a new login link was identified. Each line inside the figure is a different response from a single command.

The generation of the URL was identified in `DVRIPConfigManager.cpp` and namespace `Dahua::DVRIP::CDVRIPConfigManager::getURLInfo`.

```
b'0::0::0::0'
b'2.800.0000004.0.R'
b'Dahua::'
b'English|Italian|Spanish|Russian|French|German|Portuguese|Polish|Korean|Farsi|Czech|Dutch|
Arabic|SpanishEU&&English'
b'FTP:1:Record,Snap&&SMTP:1:AlarmT SNIP
b'http://192.168.178.108:80/config/index.htm?dXNlcm5hbWU9YWRtaW4mcGFzc3dvcmQ9MjJBMkZCMEFDRT
YxNUEzNDRGMzIyMTJBNUJCQjVFNjc5NjNEODI4Q0RBQTg1MDJDM0U2NThCRjUyMzVEQ0M0RA=='
b'IPC-HFW1431S'
b'SNAP&1&0::SIZE:18:19:17:0:5:3::FREQUENCE:1:-2:-3:-4:-5:-6:-
7::MODE:0:1::FORMAT:1::QUALITY:100:80:60:50:30:10'
      5 binaryout is: b'\x00\x00\x00\x00'
```

Figure 43: Response summary of binary engine command search

7.4 FUZZING RPC OVER DVRIP

Note that the identified DVRIP behaviour and byte sequences are based on the results from the protocol identification process shown in the result section 8.

7.4.1 CASE 6: COMMAND INJECTION IN RPC CONSOLE

Fuzzing Endpoint

In this test case, the fuzzing endpoint is the system management console, also called debug console. The console can be accessed via valid username and password over the DVRIP protocol.

Manipulation and expected results

In this fuzzing case, a manipulation of RPC commands (runCMD parameter) is used to try to trigger some errors on the target device or execute code on the target system. Figure 44 shows the manipulated header fields. Besides the overall expected results, it is expected to detect some unknown behaviour of the protocol.

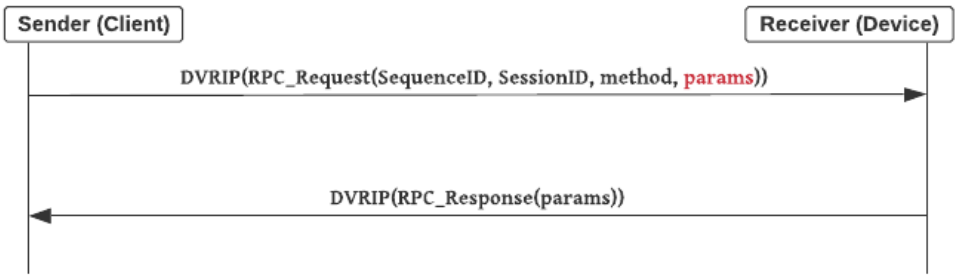


Figure 44: Sequence diagram with manipulated fields

Implementation details

According to Figure 44, the fields are manipulated and send to the target device. The implementation contains a typical command injection test and also a testing for buffer overflow vulnerabilities. Figure 45 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script.

```
        with open(filepath) as fp:
            for cnt, line in enumerate(fp):
                time.sleep(2)
                logging.info("Line {}: {}".format(cnt, line))
                buf = line
                buf2 = buf
                if injectionType == "bof":
                    buf2 = eval(buf)
                #print(buf2)

                api_argsFuzz = {
                "id":       dvripSession.sequenceID,
                "magic":    "0x1234",
                "method":   "console.runCmd",
                "params": {
                    #"command": line.rstrip('\n')
                    "command": buf2
                },
                "object":   dvripSession.objectID,
                "SID":      dvripSession.serviceID,
                "session":  dvripSession.sessionID,
                }

                data = json.dumps(api_argsFuzz)

                dvripSession.callJsanAPIFUZZ(data)

                dvripSession.ConsoleResult(dvripSession.DVRIP_PDU_response.b9_payload)
                logging.debug("next command OS-Injection fuzz payload")
```

Figure 45: Implementation details of fuzzing case 6

Observations and results

The LogEngine did not return any useful data. It seems that the target service (memory command) is no longer implemented.

7.4.2 CASE 7: FUZZING RPC CONSOLE: MEMORY COMMAND

Fuzzing Endpoint

In this test case, the fuzzing endpoint is the system management console, also called debug console. The console can be accessed via valid username and password over the DVRIP protocol. In this debug console a command exists to manipulate the device memory. This memory command is the fuzzing target of this use-case.

Manipulation and expected results

In this fuzzing case, a manipulation of RPC commands (runCMD parameter) is used to try to trigger some errors on the target device or execute code on the target system. Figure 46 shows the manipulated header fields. Besides the overall expected results, it is expected to detect some unknown behaviour of the protocol.

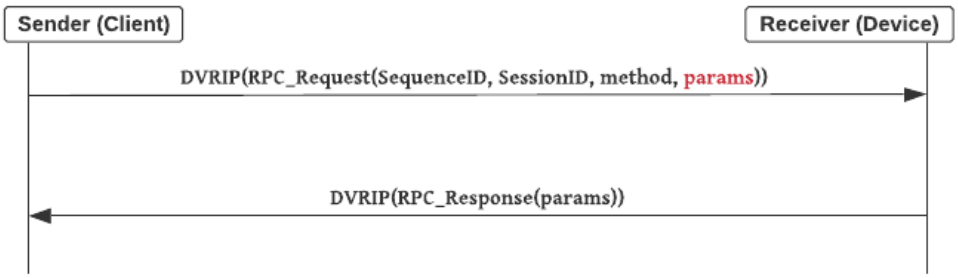


Figure 46: Sequence diagram with manipulated fields

Implementation details

According to Figure 46, the fields are manipulated and sent to the target device. Figure 47 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script.

```
api_argsFuzz = {
            "id":       dvripSession.sequenceID,
            "magic":    "0x1234",
            "method":   "console.runCmd",
            "params": {
                "command": "memory -a '0x" + binascii.hexlify(os.urandom(4)).decode() + "'"
            },
            "object":   dvripSession.objectID,
            "SID":      dvripSession.serviceID,
            "session":  dvripSession.sessionID,
```

```
}
```

Figure 47: Implementation details of fuzzing case 7

Observations and results

The LogEngine did not return any useful data and no possibility was identified to use the memory command from the console. It is not known if the memory service is really working on the target device. A fortunate usage of the memory command would result in an arbitrary manipulation of the target device and could, therefore, be disabled as a precaution to protect the target system.

7.5 FUZZING RPC OVER DHIP UDP DISCOVERY

Note that the identified DVRIP behaviour and byte sequences are based on the results from the protocol identification process shown in the result section 8.

7.5.1 CASE 8: EXTENDED FUZZING DISCOVERY, SEARCH FOR METHODS

Fuzzing Endpoint

As fuzzing endpoint, the RPC Engine on the target device is in focus. The endpoint was detected and implemented with the usage of the demo SDK applications, mentioned in subsection 5.3.2. The configuration tools from Dahua and the particular SDK demo application has a function to discover new devices in the current subnet. These requests are made over the DHIP protocol. The DHIP protocol is mentioned in section 3 and also discovered in the reverse engineering process, noted in subsection 6.4 DHIP header format.

Manipulation and expected results

In this fuzzing case the allowed RPC calls on that endpoint are in focus. In general, there is always some authorisation which is needed and tested in the following case. Figure 48 shows the manipulated header fields. Besides the overall expected results, it is expected to detect some unknown behaviour of the protocol.

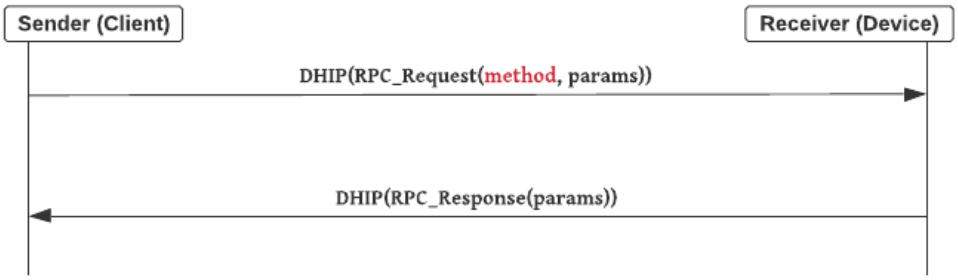


Figure 48: Sequence diagram with manipulated fields

Implementation details

According to Figure 48, the RPC methods that are manipulated and sent to the target device. In focus are the second 4 bytes of the DVRIP packet. Figure 49 shows the python implementation details for that purpose. See [22] for the full Fuzzing Engine Script. The RPC services and methods are collected with [18].

```
filepath = '/root/Master/fuzzing/lists/services.dump.list'
    with open(filepath) as fp:
        for cnt, line in enumerate(fp):
            logging.debug("Line {}: {}".format(cnt, line))

            api_argsFuzz = {
            #"method": line.rstrip('\n'),
            "method": line.rstrip('\n'),
            "params": {
                "mac": "",
                "uni": 1
                }
            }

            data = json.dumps(api_argsFuzz)

            payloadLength = len(data) #84 (int) => 54 in hex; hex(84):54
            payloadLength = struct.pack("<I", payloadLength)
            payloadLength = binascii.hexlify(payloadLength)


            udpPack1 = DVRIP_PDU(
                b'20000000', b'44484950',
                b'00000000', b'00000000',

                payloadLength, b'00000000',
                payloadLength, b'00000000',

                data )

            stream = udpPack1.stream()


            UDP_IP = "192.168.178.108"

            UDP_PORT = 37810
            #MESSAGE = searchDeviceStream
```

```
            MESSAGE = stream

            logging.debug("UDP target IP: %s", UDP_IP)
            logging.debug("UDP target port: %s", UDP_PORT)
            #logging.debug "message:", MESSAGE

            sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            #my_bytes = binascii.a2b_hex(MESSAGE)
            sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
            try:
                sock.settimeout(0.9)
                data = sock.recv(1024)
                print(data)
                print("Line {}: {}".format(cnt, line))

            except socket.timeout as e:
                logging.debug(e)
                sys.stdout.write("*")
                counter1 +=1
                logging.debug("next")
```

Figure 49: Implementation details of fuzzing case 8

Observations and results

In this analysis, it was observed that some of the RPC methods are available without any proper session identifier or authorisation. The identified methods are DHDiscover.search (default), deviceDiscovery.refresh and deviceDiscovery.ipScan. There were no vulnerabilities detected in combination with the newly gained available RPC methods.

8 RESULTS

In this section, we describe the main results of this research. The first subsection 8.1 shows the identified protocol structure, followed by subsection 8.2 with the protocol design and security mechanisms. Subsection 8.3 summarises and describes the identified vulnerabilities during this research.

8.1 DVRIP HEADER FORMAT

Figure 50 shows the identified DVRIP protocol header format. The naming of the byte blocks is based on the identified usage of that block or on the assumption of the usage of that block.

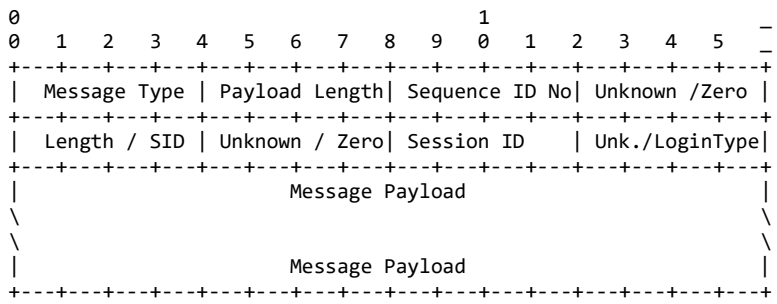


Figure 50: DVRIP protocol header format

Note that each tick mark represents a one-byte position.

Message Type: 4 bytes in length

The Message Type field describes the type of message which is used in the protocol communication. The following primary Message Types are identified:

- `A0000000` for sender login communication
- `a4000000` for sender communication with binary commands
- `F6000000` for sender communication with the RCP API

Payload Length: 4 bytes in length

The Payload Length field contains the length in little-endian from the Message Payload if existent.

Sequence ID No: 4 bytes in length

The Sequence ID No field contains either a binary command ID or the sequence number of the communication, increased each request-response by one.

Unknown / Zero1: 4 bytes in length

This field sometimes contains only zeros or arbitrary content. Could not be declared during this research.

Length / SID: 4 bytes in length

This field mostly contains the length of the payload if it is a sender message. It contains the session identifier if it is used in the login process from the device.

Unknown / Zero2: 4 bytes in length

This filed sometimes contains only zeros or arbitrary content. Could not be declared during this research.

Session ID: 4 bytes in length

Contains the session identifier when sending authenticated requests from the sender side. This session identifier is also used as an integer value in the payload field when issuing RPC calls.

Unknown / Login Type: 4 bytes in length

This field contains arbitrary content or some identifier type values when used during the login process.

Message Payload: mutable length

This field is optional. It contains either binary content or most of the time RPC commands in JSON format. The main commands of the overall communication are stored in the message payload.

8.2 IDENTIFIED PROTOCOL USE CASES

8.2.1 USE CASE 1: LOGIN WITH A 4-WAY HANDSHAKE

As we can see in the byte stream shown in subsection 5.4 Figure 13, the client is authenticated using a 4-way login handshake and receives a valid session identifier when using valid login credentials. Figure 51 shows the sequence diagram of the 4-way login handshake.

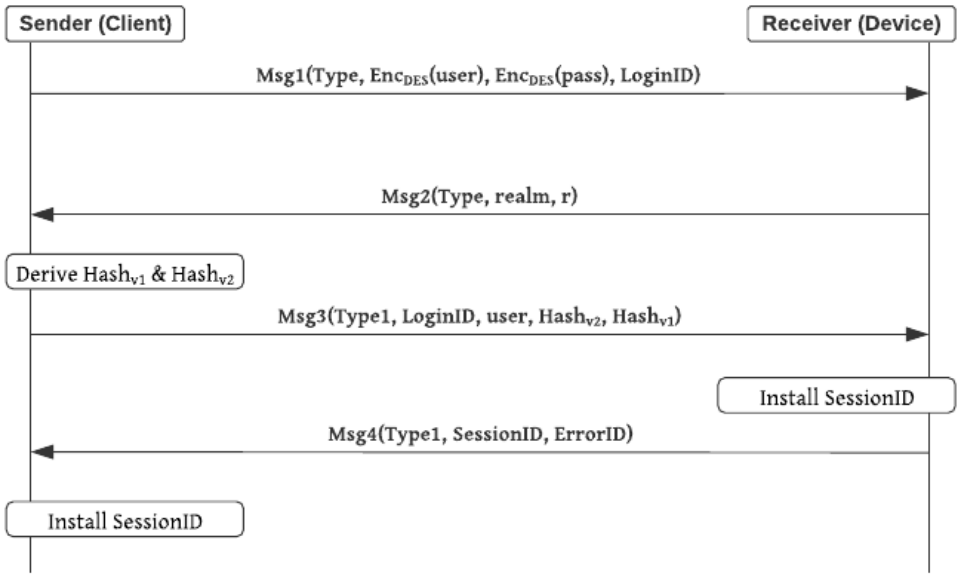


Figure 51: 4-way login handshake sequence diagram

The login 4-way handshake is initiated on the sender side with Msg1. This is defined via the type field. In Msg1 a kind of LoginID is also sent. Likewise, the username and password, presumably only up to a maximum length of 8 characters, are sent using DES encryption. However, these two ciphersuites are discarded in the 4-way login handshake on the device side. Instead, the answer is Msg2. This again contains the type of the DVRIP message and also a realm, a kind of serial number, and a random number r. On the transmitter side, two hash sums are now formed from the values obtained from Msg2. Hashv1 is a fall-back hash method that is probably only used on older devices and Hashv2 is a newer method that consists of calculating MD5 sums. In Msg3, the DVRIP type, the

LoginID and additionally a new username and the two hashes are sent from the sender side. The device now checks the hash sums and returns an active SessionID in Msg4 if valid. This session ID is now valid on the device side. Figure 52 shows the byte sequence of that 4-way login handshake.

```
Message 1                                   Message 2
Dahua DVRIP Protocol                        Dahua DVRIP Protocol
    b1 - Message Type        : 0xa0050060       b1 - Message Type        : 0xb0000058
    b2 - Payload Length      : 0x0000           b2 - Payload Length      : 0x35000000
    b3 - Sequence ID         : 0xc4a3af48       b3 - Sequence ID         : 0x10e0109
    b4 - Unknown             : 0x9956b6b4       b4 - Unknown             : 0x33000000
    b5 - Payload Length / SID: 0xfa269dec       b5 - Payload Length / SID: 0x0000
    b6 - Unknwon             : 0x29d84afb       b6 - Unknwon             : 0x1881300
    b7 - SessionID           : 0x5020001        b7 - SessionID           : 0x600f900
    b8 - Zero / LoginType    : 0xa1aa           b8 - Zero / LoginType    : 0x16402
                                                b9 - Payloaod             : Realm:Login
                                               to 803906081050579 Random:1384860226


Message 3                                   Message 4
Dahua DVRIP Protocol                        Dahua DVRIP Protocol
    b1 - Message Type        : 0xa0050060       b1 - Message Type        : 0xb0000058
    b2 - Payload Length      : 0x47000000       b2 - Payload Length      : 0x0000
    b3 - Sequence ID         : 0x0000           b3 - Sequence ID         : 0x80109
    b4 - Unknown             : 0x0000           b4 - Unknown             : 0x33000000
    b5 - Payload Length / SID: 0x0000           b5 - Payload Length / SID: 0xb8212f6e
    b6 - Unknwon             : 0x0000           b6 - Unknwon             : 0x1881300
    b7 - SessionID           : 0x5020008        b7 - SessionID           : 0x600f900
    b8 - Zero / LoginType    : 0xa1aa           b8 - Zero / LoginType    : 0x16402
    b9 - Payload             : admin&&
37A66693916B0B2B066EE21E4654A1D1
B80D8BFE4D0AC1AE5FFA09A34E3F4464
```

Figure 52: Byte sequence of the 4-way login handshake 1

8.2.2 USE CASE 2: LOGIN WITH A 2-WAY HANDSHAKE (DES FALLBACK)

As part of the analysis, it was also found that with the 4-way login handshake, username and password are transmitted as a DES-encrypted cipherstream in the first message. This was identified in the later research, described in subsection 7.3.2. The message flow is illustrated in Figure 53. The sender (client) sends the DVRIP message type in Msg1 and the username and password encrypted with DES. Based on the block size, a username and password of up to 8 characters is probably possible. The login ID is adjusted so that the receiver (device) performs direct authentication. If the decryption and authentication are successful a new active session ID is generated.

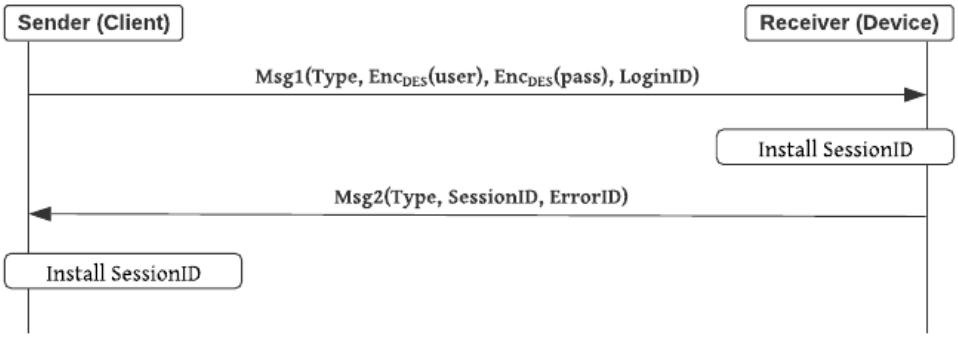


Figure 53: Login with DES encryption – 2-way handshake

Figure 54 shows the byte sequence of the 2-way DES login handshake.

```
Message 1                                  Message 2
Dahua DVRIP Protocol                       Dahua DVRIP Protocol
    b1 - Message Type        : 0xa0745f26      b1 - Message Type        : 0xb0000058
    b2 - Payload Length      : 0x0000          b2 - Payload Length      : 0x0000
    b3 - Sequence ID         : 0xc4a3af48      b3 - Sequence ID         : 0x80109
    b4 - Unknown             : 0x9956b6b4      b4 - Unknown             : 0x33000000
    b5 - Payload Length / SID: 0x6e6302f2      b5 - Payload Length / SID: 0x7eac9e6d
    b6 - Unknwon             : 0xb792f12c      b6 - Unknwon             : 0x1881300
    b7 - SessionID           : 0x5020001       b7 - SessionID           : 0x600f900
    b8 - Zero / LoginType    : 0x0000          b8 - Zero / LoginType    : 0x16402
```

Figure 54: Byte sequence of the 2-way handshake login with DES encryption

8.2.3 USE CASE 3: EXECUTING BINARY COMMANDS

As part of the analysis, it was identified that commands could be sent via the DVRIP binary header. In this way, the following commands were identified as byte blocks:

- `'08000000'` for querying the software version
  `'2.800.0000004.0.R'`
- `'07000000'` for querying the serial number
  `'5C0726FPAGEC7C2'`
- `'0b000000'` for querying the model type
  `'IPC-HFW1431S'`

The command IDs are defined in the third-byte block, named CommandID. If the TCP connection is authenticated, the corresponding results of the command are returned, see subsection 8.3.2 for details about the authenticated TCP connection and Figure 55 as an overview of the binary command sequence diagram.

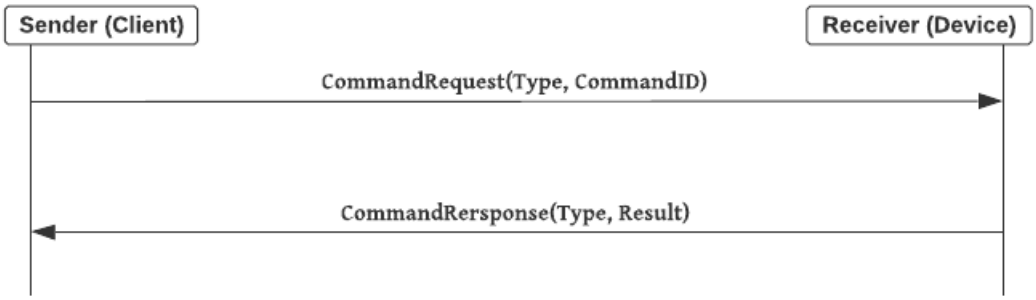


Figure 55: Binary command sequence diagram

Figure 56 shows how the DVRIP packets were sent:

```
Message 1                                Message 2(4)
Dahua DVRIP Protocol                     Dahua DVRIP Protocol
    b1 - Message Type        : 0xa4000000        b1 - Message Type        : 0xb4000058
    b2 - Payload Length      : 0x0000            b2 - Payload Length      : 0xf000000
    b3 - Sequence ID         : 0x8000000         b3 - Sequence ID         : 0x8000000
    b4 - Unknown             : 0x0000            b4 - Unknown             : 0x0000
    b5 - Payload Length / SID: 0x0000            b5 - Payload Length / SID: 0x0000
    b6 - Unknwon             : 0x0000            b6 - Unknwon             : 0x0000
    b7 - SessionID           : 0x0000            b7 - SessionID           : 0x0000
    b8 - Zero / LoginType    : 0x0000            b8 - Zero / LoginType    : 0x0000
                                                 b9 – Payload             : 2.400.BW01.3.R
```

Figure 56: Implementation details of fuzzing case 1

8.2.4 USE CASE 4: EXECUTING COMMANDS ON THE RPC SERVER

Another option for executing commands on the device is querying the RPC Server on the target device over an API. This is addressed via the DVRIP protocol and called up using JSON syntax. The RPC calls can partly be called up directly and partly via an extended service sequence. Figure 57 shows the simple RPC call. The RPC command can directly be sent, see Figure 58 as an example inside an authenticated session.

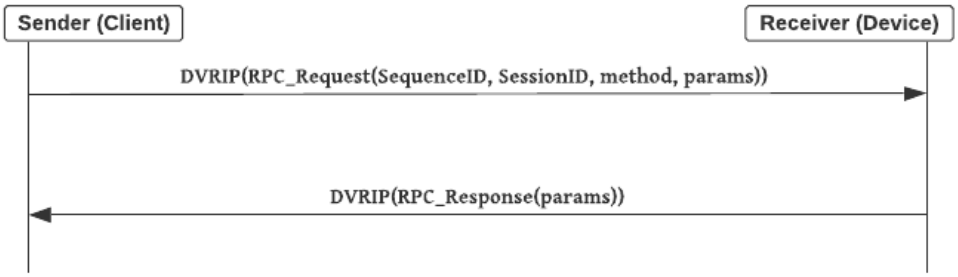


Figure 57: DVRIP RPC communication

```
JSON_RPC_Request
api_args = {
    "method":"magicBox.getSerialNo",
    "params": None,
    "session": self.sessionID,
    "id": self.sequenceID,
    }
```

```
JSON_RPC_Response
response = {
    "id" : 2,
    "params" : {
      "type" : "BURGcamBULLET" },
      "result" : true,
      "session" : 1166699139
    }
```

Figure 58: Byte sequence of short RPC calls

Figure 59 illustrates the process with an extended service sequence. An active session is required to access the RPC API, e.g. with the mentioned 4-way login handshake. The JSON-RPC communication is embedded in a DVRIP header. This header is omitted in this figure for a better overall view. First, 'SERVICE.factory.instance' is called for the desired service. This RPC generates an ObjectID on the device, which is assigned to the service and returned. After that, the desired service for the current session is assigned or "attached". For this purpose, a ServiceID is assigned on the device side and returned to the client. As the last step, with knowledge of an ObjectID and ServiceID, the target service can now be fed with commands. This step can be repeated several times.

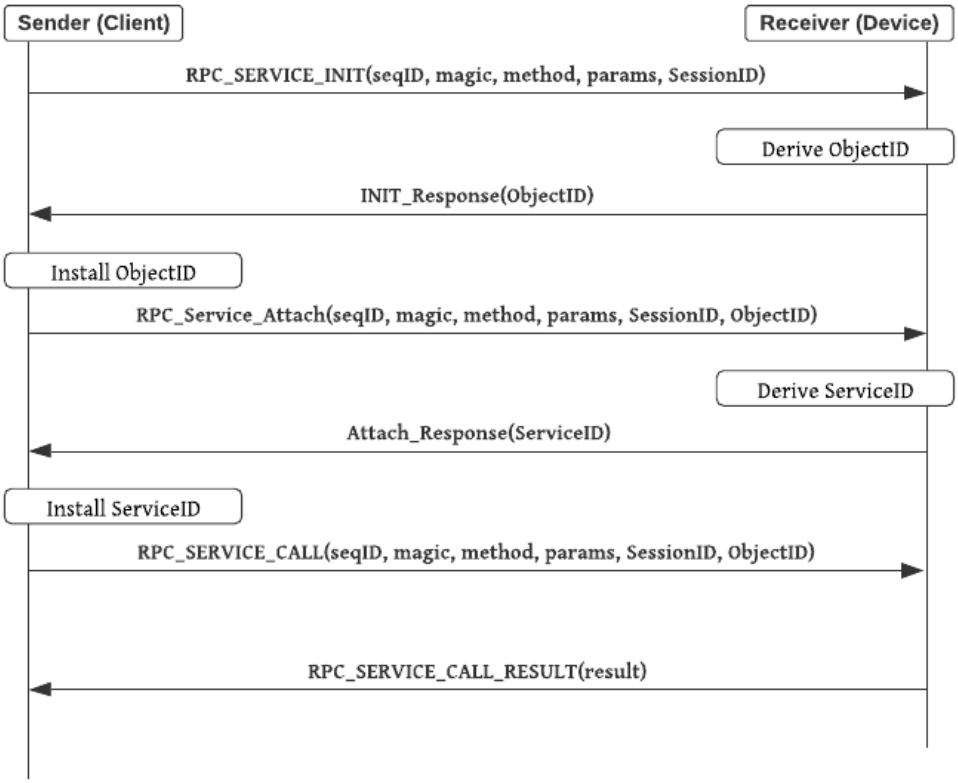


Figure 59: DVRIP RPC communication with console service

Figure 60 shows the byte sequence of extended RPC calls.

```
JSON_RPC_Request
api_args = {
    "id": 5,
    "magic":
    "0x1234",
    "method": "console.factory.instance",
    "params": null,
    "session": 1166699139}
```

```
JSON_RPC_Response
response = {
    "id" : 5,
    "params" : null,
    "result" : 40093192,
    "session" : 1166699139 }
```

```
JSON_RPC_Request
api_args = {
    "id": 6,
    "magic": "0x1234",
    "method": "console.attach",
    "params": {"proc": 6},
    "object": 40093192,
    "session": 1166699139}
```

```
JSON_RPC_Response
response = {
    "id" : 6,
    "params" : { "SID" : 6 },
    "result" : true,
    "session" : 1166699139 }
```

```
JSON_RPC_Request
api_args = {
    "id": 7,
    "magic": "0x1234",
    "method": "console.runCmd",
    "params": {"command": "user"},
    "object": 40093192,
    "SID": 6,
    "session": 1166699139}
```

```
JSON_RPC_Response
response = {
    "id" : 6,
    "method" : "client.notifyConsoleResult",
    "object" : 40093192,
    "params" : {
      "SID" : 6,
      "info" : {
         "Count" : 4,
         "Data" : [ "DATA DATA DATA \n" ] }
},
    "session" : 1166699139 }
```

Figure 60: Byte sequence of extended RPC call

8.2.5 USE CASE 5: LOGOUT WITH A 2-WAY HANDSHAKE

A session is terminated with a logout request from the client-side. To do this, the client sends a DVRIP packet with the corresponding message type and the active session ID, see Figure 61.

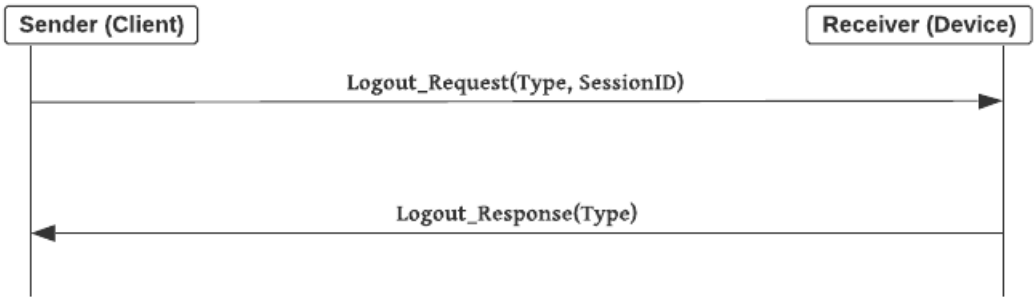


Figure 61: DVRIP logout flow graph

The corresponding byte sequence is shown in Figure 62.

```
Message 1                                   Message 2(4)
Dahua DVRIP Protocol                        Dahua DVRIP Protocol
    b1 - Message Type        : 0xa000000        b1 - Message Type        : 0xb010078
    b2 - Payload Length      : 0x0000           b2 - Payload Length      : 0x0000
    b3 - Sequence ID         : 0xfdffff7f       b3 - Sequence ID         : 0x0000
    b4 - Unknown             : 0x0000           b4 - Unknown             : 0x0000
    b5 - Payload Length / SID: 0x0000           b5 - Payload Length / SID: 0x0000
    b6 - Unknwon             : 0x0000           b6 - Unknwon             : 0x0000
    b7 - SessionID           : 0x0000           b7 - SessionID           : 0x0000
    b8 - Zero / LoginType    : 0x0000           b8 - Zero / LoginType    : 0x0000
```

Figure 62: Byte sequence of the logout process

8.3 PROTOCOL DESIGN AND SECURITY MECHANISMS

With the knowledge of the mechanisms and structure of the DVRIP protocol, the various security mechanisms of the protocol have now been examined. Note that on the older firmware an SSL/TLS DVRIP port was available which is not accessible anymore in the newer firmware. Theoretically, also identified in the application binary, there should be a DVRIP endpoint. The communication with DVRIP could not be tested due to the lack of the open SSL/TLS port.

Identified security mechanisms of the DVRIP Protocols:

- DES encryption for fallback login
- Session handling with TCP based authentication (4-way handshake)
- Invalid packet protection (no response with malicious packet)

8.3.1 DES ENCRYPTION FOR FALLBACK LOGIN

A fallback login process was identified as part of the analysis of the 4-way login process, described in subsection 7.3.2. Within the first message, the username and password are sent to the device encrypted by DES. The device then checks whether the message should perform a fallback login with the appropriate parameters.

8.3.2 SESSION HANDLING WITH TCP-BASED AUTHENTICATION

With the analysis of the login process, a state machine for the session handling on the target device was defined, shown in Figure 63. This state machine includes the login mechanism over the 4-way handshake and a 2-way handshake. Once the target device enters the 4-way or 2-way handshake, the state changes to LOGIN-INIT. In this state, the target device is listening for incoming messages with the correct protocol syntax and data. If a message with Type2, representing a 4-way handshake login, is incoming, the state machine changes to LOGIN-START. The Sonia core system then retrieves the REALM for the internal RPC Server and also calculates a random value for the communication. This data is sent back to the device as Msg2. Once Msg3 is retrieved from the client, containing valid credentials, the state machine changes to LOGIN-DONE. It is also possible to reach this state from LOGIN-INIT when the client sends an Msg1 with valid credentials directly as DES-encrypted data. In the state LOGIN-DONE, the device generates and activates a SessionID. The active SessionID is then sent back to the client. The client has successfully authenticated to the device and retrieved a valid SessionID. Once a fifth DVRIP message or protocol data unit (PDU) is retrieved from the same client, the state machine changes to LOGIN-BIND. Then the active SessionID is bonded to the active TCP session.

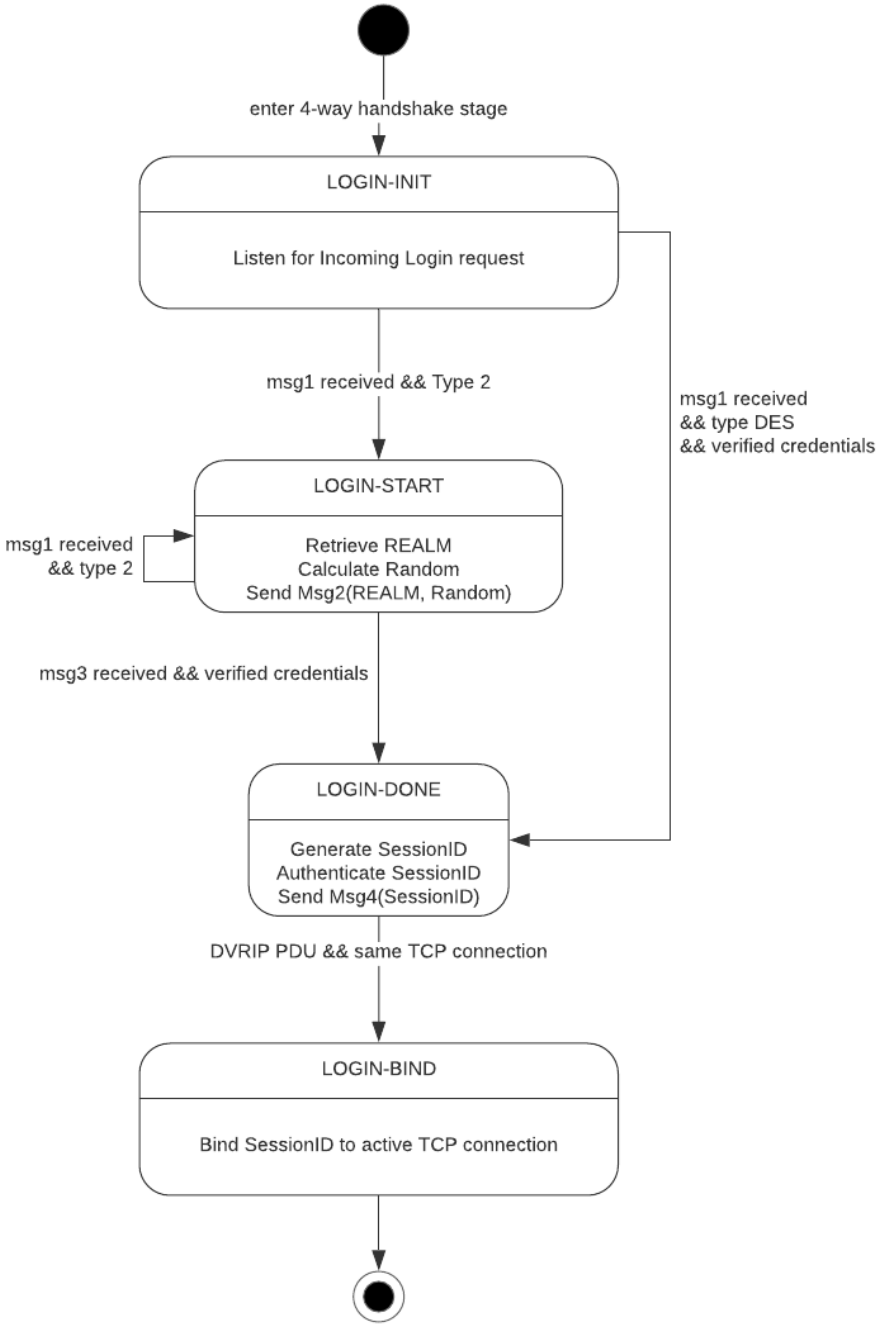


Figure 63: Session handling login state machine

8.3.3 INVALID PACKET PROTECTION

If there is no defined state within the application for a DVRIP packet, no response is sent back to the client. This makes various attacks very difficult. As part of the analysis, however, the logging engine could be used, which made the later analysis via fuzzing easier and increased the quality of the results. The invalid packet protection mechanism was identified in the following scenarios:

- Packets containing an invalid or inactive session identifier
- Packets that contain non-existing RPC services or methods
- Packets that contain malicious payload

8.4 IDENTIFIED VULNERABILITIES

With the knowledge of the mechanisms and structure of the DVRIP protocol, the various security mechanisms and the general design of the protocol were examined. Based on the defined security requirements for secure protocols in subsection 2.4, the DVRIP protocol is considered and summarised in Figure 64.

Security requirement	How it is met
Data confidentiality	No security mechanism detected
Data integrity	TCP stream-based session protection
Server authentication	No security mechanism detected
Client authentication	Username/Password and other optional login-types

Figure 64: DVRIP security requirements rating

Identified (design) vulnerabilities:

- 'Allways on' DES fallback login function
- Usage of deprecated DES encryption
- Bypass of TCP based authentication
- CVE-2020-9502: Session Hijacking / Brute-Force

8.4.1 „ALLWAYS ON“ DES FALLBACK FUNCTION AND MISSING MAN-IN-THE-MIDDLE PROTECTION

The SDK and the ConfigTool send the DES credentials in standard use cases. The credentials are always sent even though they are rejected on the device side by Msg1. If an attacker can record this network communication, the attacker gets access to the encrypted credentials and can replay them. This vulnerability was also discovered from Bashis and published in early May 2020 [33].

8.4.2 USAGE OF DEPRECATED DES ENCRYPTION

DES encryption is out of date and can therefore be cracked. With the support of David Hulton, founder, chairman, program coordinator of the ToorCon Information Security Conference and the crack.sh service offered by ToorCon Inc, a full brute force of the DES encryption keyspace was made in about 32 hours runtime. The service describes itself as:

> *The World's Fastest DES Cracker*
>
> *In 1998 the Electronic Frontier Foundation built the EFF DES Cracker. It cost around $250,000 and involved making 1,856 custom chips and 29 circuit boards, all housed in 6 chassis, and took around 9 days to exhaust the keyspace. Today, with the advent of Field Programmable Gate Arrays (FPGAs), we've built a system with 48 Virtex-6 LX240Ts which can exhaust the keyspace in around 26 hours, and have provided it for the research community to use. Our hope is that this will better demonstrate the insecurity of DES and move people to adopt more secure modern encryption standards.*
>
> *The Technology*
>
> *Behind crack.sh is a system with 48 Xilinx Virtex-6 LX240T FPGAs. Each FPGA contains a design with 40 fully pipelined DES cores running at 400MHz for a total of 16,000,000,000 keys/sec per FPGA, or 768,000,000,000 keys/sec for the whole system. This means that it can exhaustively search the entire 56-bit DES keyspace in:* 2^{56} */ 768,000,000,000 = ~26 hours [34]*

A comparison of other cracking systems can be found in summary by Sugier [35, p. 481]. See subsection 7.3.4 how the possible keys were tested on the target system.

8.4.3 BYPASS OF TCP BASED AUTHENTICATION

As part of the fuzzing analysis, it was identified that the binding process of an authenticated SessionID to a TCP connection can be circumvented indirectly. Figure 65 illustrates the state machine of login session handling on the device. The status LOGIN-DONE of a new TCP connection with the correct SessionID can be abused with a new TCP connection. The TCP connection from the 4-way handshake (or 2-way) can then no longer be used and becomes invalid. The client from the initial connection has to authenticate again. The message in LOGIN-DONE status can be repeated any number of times until a package that is valid for LOGIN-DONE is available. This means that a SessionID brute force is made possible and the TCP session binding can be bypassed indirectly. This attack scenario and vulnerability is illustrated in the next subsection 8.4.4. The vulnerability got CVE-2020-9502 as an identifier.

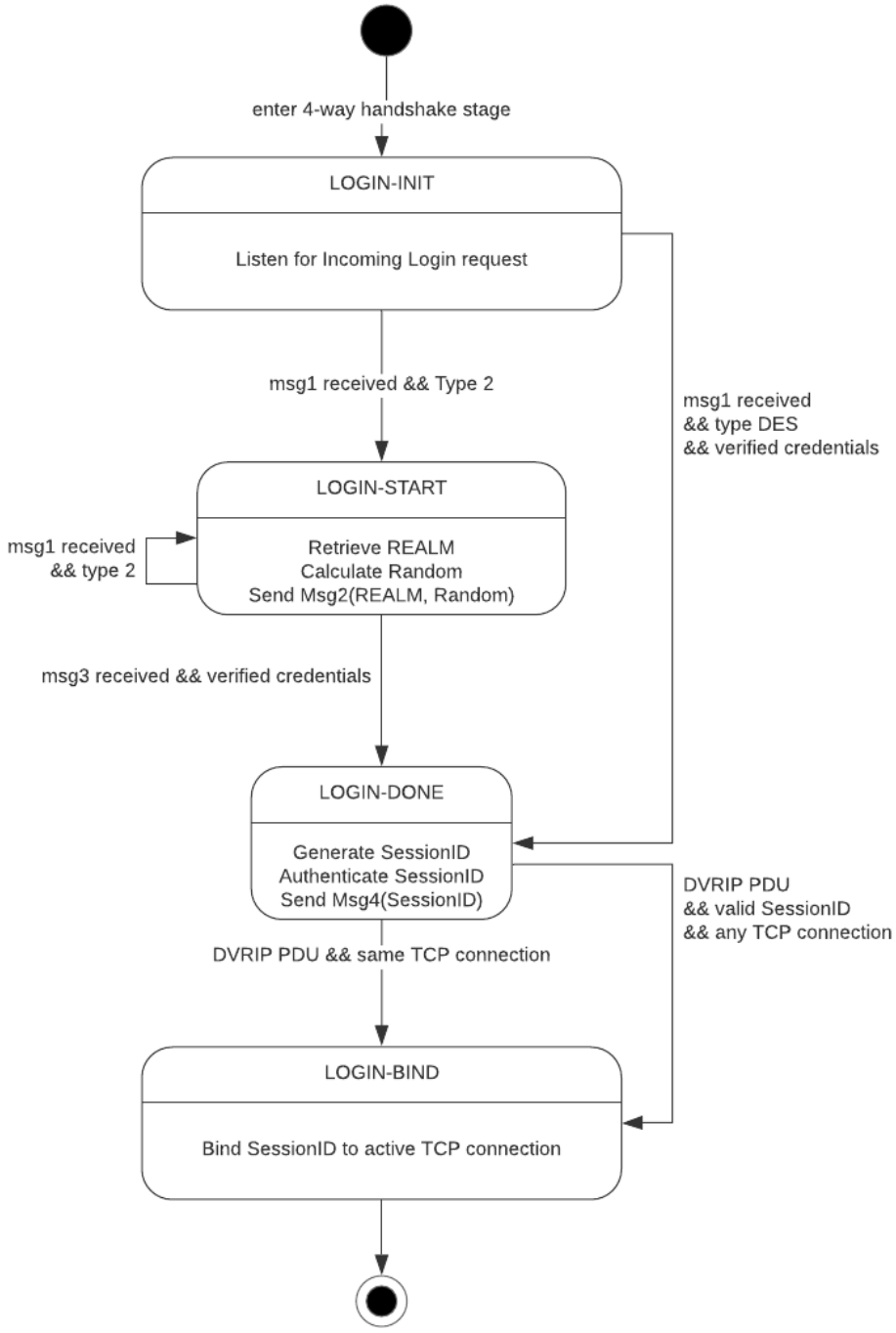


Figure 65: Session handling login state machine, TCP stream bypass

8.4.4 CVE-2020-9502: SESSION HIJACKING SECURITY ADVISORY

With the analysis of the session handling, it was identified that the generation of the session identifier contains a randomness vulnerability which can lead to a session hijacking of the target device. The vulnerability was communicated to the Dahua Product Security Incident Response Team so that the vendor can verify and update this potentially serious vulnerability. A proof-of-concept exploit was developed and a video demonstration with the working exploit was provided to the vendor. See Appendix A6 for the communicated security advisory and data. The session hijacking is possible due to a very poor session entropy. After a successful login on the target application, the application sends an active session identifier for the client, named SessionID in this description. This SessionID is used in the communication, either as binary value or as integer (little-endian) in JSON requests. Example of SessionIDs in the binary header:

- '83ffff7f'
- '82ffff7f'
- '81ffff7f'
- '80ffff7f'
- '7fffff7f'

Figure 66 shows an example of the SessionID in a JSON request:

```
{"method": "global.keepAlive", "magic": "0x1234", "params": {"timeout": 30, "active": true},
"id": 5, "session": 2147483523}
```

Figure 66: SessionID in JSON request

The red Integer 2147483523 is the binary value of '83ffff7f' (UINT32 – little endian(DCBA))

For a successful session hijacking, an adversary must only know this SessionID value. This SessionID only changes 1 byte after each successful login on the target device. With the fact that the SessionID has almost no randomness and entropy, it is very likely to gain unauthorised access with this attack. An additional behaviour was observed, that the SessionID was always starting with the same value

when the device was rebooted. Predicting the SessionID or brute-forcing the SessionID is thus extremely easy and dangerous.

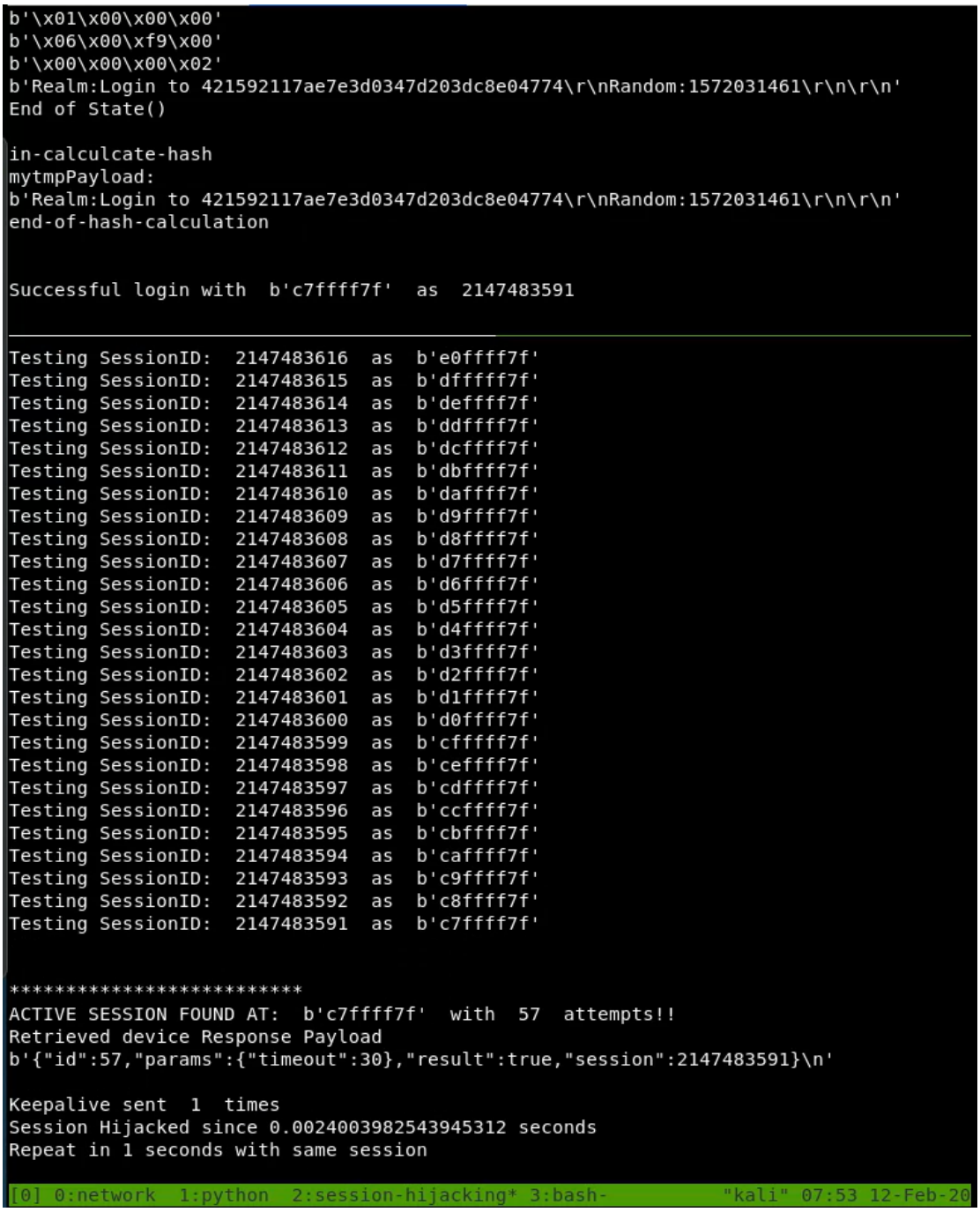


Figure 67: Running session hijacking exploit

Figure 67 shows the successful exploited session hijacking vulnerability CVE-2020-9502. First, in the upper window, a client connection is simulated. Therefore a 4-way login handshake is made so that

a valid session identifier is returned. In the lower window, the attack script is then started. The exploit script brute-forces the session identifier until it succeeds and then sends authenticated commands for testing purposes.

As part of the general vulnerability analysis, further weaknesses in connection with the analysis of the DVRIP protocol were identified and communicated with the Dahua PSIRT. These are described in the following subsections.

8.4.5 DEBUG VERSION OF MAIN SONIA BINARY:

During the research, it was discovered that the main Sonia binary are spread with debug information. This includes the official software download from Dahuas official website. With debug information inside the software, the source code could very likely be restored. Class names of the C ++ source files, C source files, function and variable names can be restored to the almost original code. Figure 68 shows the view of the decompiled Sonia binary. This allows the full recovery of over 2800 ELF Source Files (e.g. CPP and C files, e.g. `DVRIPConfigManager.cpp`) and disassembled C code from over 70.000 functions. Unprotected source code can reveal secrets, AI algorithms, passwords, encryption keys or other sensitive business logic mechanisms which can cause risks. There is also a higher probability of the existence of new exploits, based on the better understanding and reviewing the source code. Section 6 describes the details of the reverse engineering process of the Sonia binary. The existence was reported to the Dahua Product Security Incident Response Team. They removed the mentioned firmware downloads with the debug information inside. It is assumed that the debug version of the Sonia is widespread and available on various white labels [36] or third-party download sites.

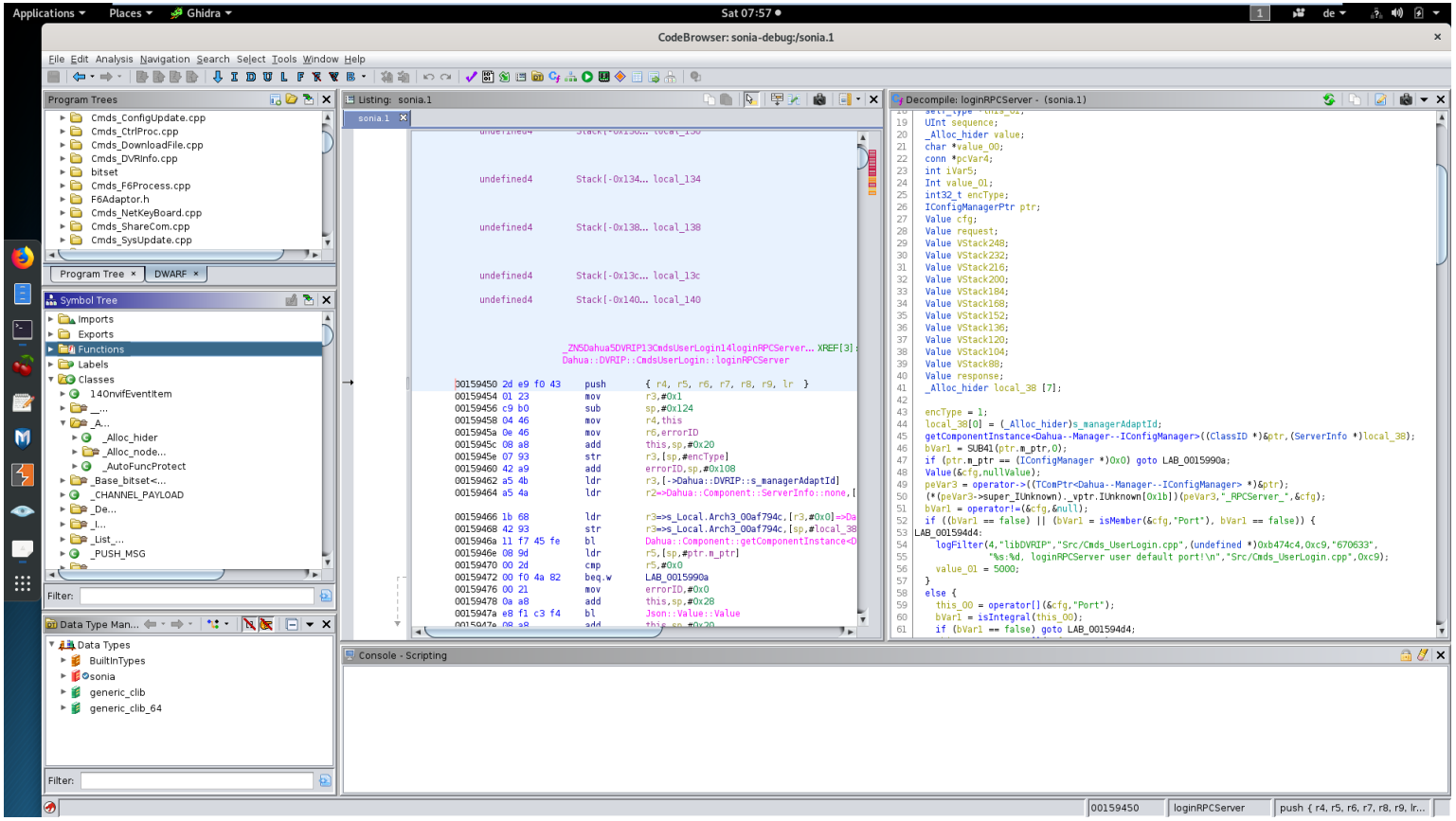


Figure 68: Decompiled Sonia binary

8.4.6 HELPDESK FIRMWARE ACCESS OF ALL DAHUA PRODUCTS:

During the research, it was also discovered that Dahua Technology is offering Helpdesk access for all of the distributed products. There were multiple official Dahua websites which revealed a username and password combination for the access. The identified websites are:

- https://dahua.support/en/support/solutions
- https://dahua.support/en/support/solutions/articles/75000018589-firmware
- https://dahua.support/en/support/solutions/articles/75000019881-firmware
- https://dahua.support/en/support/solutions/articles/75000018598-nvr-firmware
- https://dahua.support/en/support/solutions/articles/75000018607-vdp-firmware
- https://dahua.support/en/support/solutions/articles/75000018608-asc-firmware
- https://dahua.support/en/support/solutions/articles/75000018634-hdcvi-record-frimware

The website revealed a login link, a username and a valid password for Europe and a different one for China. Figure 69 shows the accessed site with download and upload possibility.

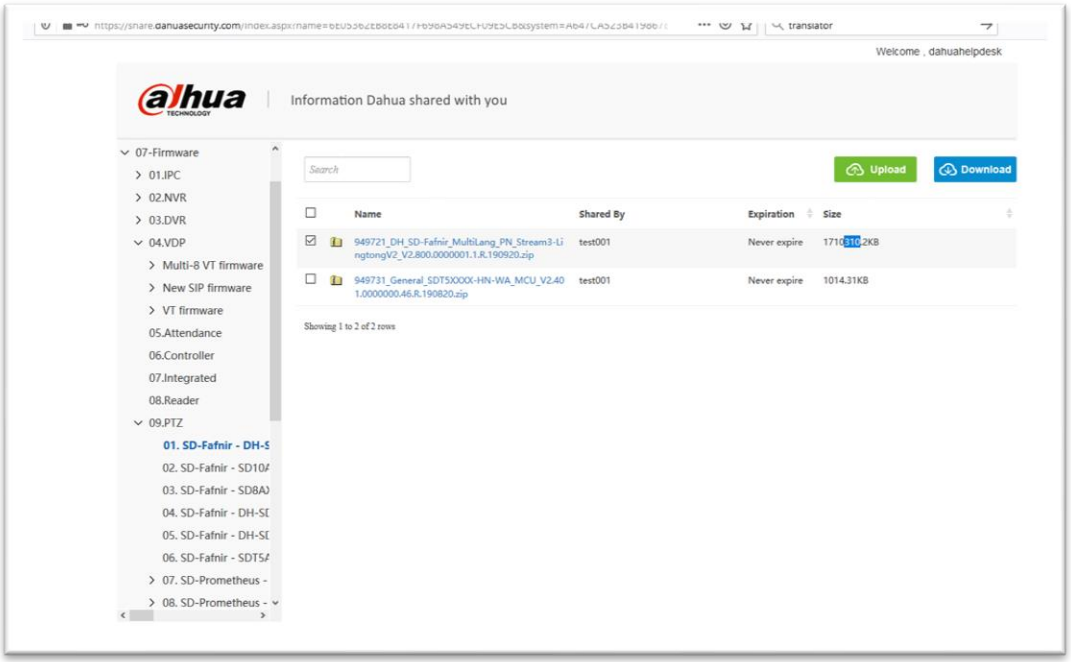


Figure 69: Access to firmware for all Dahua Technology products

The access to various firmware and potentially not officially released software, also containing debug information, could reveal sensitive data or also cause licence risks. The download functionality was verified during the research. Uploading firmware or other files was not tested due to security reasons. A working upload functionality could cause critical risk, which would allow an attacker to upload malicious software. The malicious software would then be spread from the official helpdesk users who are using the mentioned website. The Dahua Product Security Incident Response Team was informed about that potential vulnerability. It was also recommended to check the provided access whether it is valid for the public or only for internal or development usage. Dahua restricted the access after the reported vulnerability. Whether the upload functionality was working is unknown and was not commented on by Dahua. Appendix A8 contains the submitted information provided to the Dahua Product Security Incident Response Team.

9 CONCLUSION

In this research, the DVRIP protocol was examined for security-related vulnerabilities. The research goal was to identify possible vulnerabilities and to establish a research foundation on this topic. It was identified how the protocol was designed and which security mechanisms the protocol has. An analysis of the protocol identified several vulnerabilities, including CVE-2020-9502, which is a severe session hijacking vulnerability. Furthermore, general weaknesses in connection with the DVRIP protocol were identified and reported to the Dahua Product Security Incident Response Team.

Based on the research methodology, the theoretical background to the protocol reversing, both network-based and application-based, is the first element. At the start of the research, no information was available regarding the DVRIP protocol, which changed after about six weeks. The newly published information was included in the current research and, if used, marked accordingly. A network setup with suitable hardware was defined as the second element. For this purpose, the latest model of the IPC-HFW-1431S was chosen, which at that time could only be ordered directly from China. An older model was also used for the analysis, which provided additional debug interfaces. The main components of the analysis were the protocol identification, the reverse engineering of the DVRIP application, and the analysis of the DVRIP implementation.

As part of the protocol identification, various protocol mechanisms were recorded and implemented in a client. When analysing the protocol itself, it was found that a type of fallback login mechanism is active by default and secured by DES encryption. Thus, this can be attacked due to the outdated standard. It was also identified that the user session on a device is additionally secured via the active TCP session. This security mechanism can be bypassed with the identified vulnerability CVE-2020-9502, which allows a session hijacking on the target device.

Another identified vulnerability was that firmware distributed by Dahua was released using debug information. Besides that, a help desk access from Dahua was identified, which could be used via freely accessible authentication information. The firmware data stored there also contained debug information. With this combined information, the original code can be almost completely restored.

As part of the reverse engineering of the main application Sonia, this could be used to carry out more detailed analyses. An attempt was made to identify the application code and the corresponding functions. A basic functionality of the Sonia application was also defined and the DVRIP protocol was analysed in this context. The analysis of the DVRIP implementation primarily referred to the analysis of endpoints that can be used without authentication. The most significant attack vector in connection with the DVRIP protocol is that the DVRIP protocol is freely available millions of times on the Internet. A vulnerability can therefore have far-reaching effects. DVRIP-specific endpoints such as in the login process or the RPC server in the Sonia core were analysed. All functions of the ConfigTool application run primarily on it. Direct command communication (binary commands) was also analysed. In the direct binary command communication a newly identified login function on the web interface was identified, which represents new attack vectors.

The newly gained knowledge gives a great insight into the security mechanisms of the DVRIP protocol. It is assumed that, with the support of the Dahua PSIRT, an increase in the security level was achieved. In addition, the results of this research form an excellent foundation for further security analyses of the DVRIP protocol.

References

[1] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check," *IEEE Access*, vol. 7, pp. 71907–71920, 2019, doi: 10.1109/ACCESS.2019.2919760.

[2] H.-T. Nguyen, Q.-D. Ngo, and V.-H. Le, "A novel graph-based approach for IoT botnet detection," *Int. J. Inf. Secur.*, 2019, doi: 10.1007/s10207-019-00475-6.

[3] John Honovich, *Video Surveillance History.* [Online]. Available: https://ipvm.com/reports/history-video-surveillance (accessed: May 12 2020).

[4] Zhejiang Dahua Technology Co., Ltd, *Video Systems Integration: The Dahua Private Protocols Download.* [Online]. Available: https://dipp.dahuasecurity.com/integrationProtocols/112 (accessed: May 12 2020).

[5] L. Dahua Technology Co., *Dahua Technology - Leading Video Surveillance Solution Provider.* [Online]. Available: https://www.dahuasecurity.com/ (accessed: May 12 2020).

[6] shodan.io, *port:37777 - Shodan Search.* [Online]. Available: https://www.shodan.io/search?query=port%3A37777 (accessed: Nov. 1 2019).

[7] A. G. Eustis, "The Mirai Botnet and the Importance of IoT Device Security," in *Advances in Intelligent Systems and Computing, 16th International Conference on Information Technology-New Generations (ITNG 2019)*, S. Latifi, Ed., Cham, Switzerland: Springer Nature, 2019, pp. 85–89.

[8] J. Duchêne, C. Le Guernic, E. Alata, V. Nicomette, and M. Kaâniche, "State of the art of network protocol reverse engineering tools," *J Comput Virol Hack Tech*, vol. 14, no. 1, pp. 53–68, 2018, doi: 10.1007/s11416-016-0289-8.

[9] J. Forshaw, *Attacking network protocols: A hacker's guide to capture, analysis, and exploitation*. San Francisco California: No Starch Press, 2018.

[10] *OWASP Internet of Things: OWASP Internet of Things Top 10 2018.* [Online]. Available: https://owasp.org/www-project-internet-of-things/ (accessed: Apr. 6 2020).

[11] G. Bossert, "Exploiting Semantic for the Automatic Reverse Engineering of Communication Protocols," Theses, Supélec, 2014. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01146797

[12] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14*, Kyoto, Japan, 2014, pp. 51–62.

[13] James Forshaw, *tyranid/CANAPE.Core: A network proxy library written in C# for .NET Core based on CANAPE.* [Online]. Available: https://github.com/tyranid/CANAPE.Core (accessed: Apr. 7 2020).

[14] J. F. Michael Jordon, *Black Hat ® Technical Security Conference: Europe 2012 // Home: CANAPE: Bytes Your Bits.* [Online]. Available: https://www.blackhat.com/html/bh-eu-12/bh-eu-12-archives.html#CANAPE (accessed: Apr. 7 2020).

[15] mogwailabs, *mogwailabs/canape-workshop2018: CANAPE workshop at BSides Munich/Lisbon 2018* (accessed: Apr. 7 2020).

[16] bashis, *mcw0/PoC: Exploit and Script Collection.* [Online]. Available: https://github.com/mcw0/PoC/ (accessed: Apr. 7 2020).

[17] bashis, *Dahua-DHIP-JSON-Debug-Console.py.* [Online]. Available: https://github.com/mcw0/PoC/blob/master/Dahua-DHIP-JSON-Debug-Console.py (accessed: Nov. 1 2019).

[18] bashis, *Dahua-JSON-Debug-Console-v2.py.* [Online]. Available: https://github.com/mcw0/Tools/blob/master/Dahua-JSON-Debug-Console-v2.py (accessed: Mar. 11 2020).

[19] The Tcpdump Group, *TCPDUMP/LIBPCAP public repository.* [Online]. Available: https://www.tcpdump.org/ (accessed: Apr. 14 2020).

[20] wireshark.org, *Wireshark · Go Deep.* [Online]. Available: https://www.wireshark.org/ (accessed: Apr. 14 2020).

[21] *Lua/Dissectors - The Wireshark Wiki.* [Online]. Available: https://wiki.wireshark.org/Lua/Dissectors (accessed: Mar. 23 2020).

[22] Thomas Vogt, *r4bit999/dvrip-analysis.* [Online]. Available: https://github.com/r4bit999/dvrip-analysis/tree/thesis (accessed: Mar. 5 2020).

[23] L. Dahua Technology Co., *Tools: Maintenance Tools.* [Online]. Available: https://www.dahuasecurity.com/support/downloadCenter/tools (accessed: Mar. 11 2020).

[24] L. Dahua Technology Co., *Software: SDK.* [Online]. Available: https://www.dahuasecurity.com/support/downloadCenter/softwares?child=3 (accessed: Mar. 11 2020).

[25] F. G. Georges Bossert, *GitHub - netzob/netzob: Netzob: Protocol Reverse Engineering, Modeling and Fuzzing.* [Online]. Available: https://github.com/netzob/netzob (accessed: Mar. 11 2020).

[26] Georges Bossert, *Update Readme for external references, tutorials by r4bit999 · Pull Request #159 · netzob/netzob.* [Online]. Available: https://github.com/netzob/netzob/pull/159/files#diff-88b99bb28683bd5b7e3a204826ead112 (accessed: Mar. 25 2020).

[27] *CVE - CVE-2017-8229.* [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8229 (accessed: Apr. 1 2020).

[28] *CVE - CVE-2017-8230.* [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8230 (accessed: Apr. 1 2020).

[29] *CVE - Search Results.* [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=dahua (accessed: Apr. 1 2020).

[30] *Ghidra.* [Online]. Available: https://ghidra-sre.org/ (accessed: Apr. 1 2020).

[31] Microsoft, *Namespaces (C++).* [Online]. Available: https://docs.microsoft.com/en-gb/cpp/cpp/namespaces-cpp?view=vs-2019 (accessed: Apr. 3 2020).

[32] crack.sh, *h1kari/des_kpt.* [Online]. Available: https://github.com/h1kari/des_kpt (accessed: Apr. 10 2020).

[33] bashis, *Dahua-3DES-IMOU-PoC.py.* [Online]. Available: https://github.com/mcw0/PoC/blob/master/Dahua-3DES-IMOU-PoC.py (accessed: May 11 2020).

[34] ToorCon Inc., *crack.sh | The World's Fastest DES Cracker.* [Online]. Available: https://crack.sh/ (accessed: Apr. 7 2020).

[35] J. Sugier, "Cracking the DES Cipher with Cost-Optimized FPGA Devices," in *Advances in intelligent systems and computing, 2194-5365*, volume 987, *Engineering in dependability of computer systems and networks: Proceedings of the fourteenth International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, July 1--5, 2019, Brunów, Poland / Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak and Janusz Kacprzyk*, W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, Eds., Cham, Switzerland: Springer, 2020, pp. 478–487.

[36] IPVM, *Dahua OEM Directory.* [Online]. Available: https://ipvm.com/reports/dahua-oem (accessed: May 5 2020).

A APPENDIX

This section contains the appendix of the thesis.

A1 GHIDRA IMPORT SUMMARY

```
Project File Name:      sonia
Last Modified:  Thu Feb 20 23:18:21 CET 2020
Readonly:       false
Program Name:   sonia
Language ID:    ARM:LE:32:v8 (1.102)
Compiler ID:    default
Processor:      ARM
Endian: Little
Address Size:   32
Minimum Address:00010000
Maximum Address:_elfSectionHeaders::00000667
# of Bytes:     230519338
# of Memory Blocks:     43
# of Instructions:      4
# of Defined Data:      211494
# of Functions: 70198
# of Symbols:   107864
# of Data Types:462
# of Data Type Categories:      2
Created With Ghidra Version:    9.1.1
Date Created:   Thu Feb 20 23:16:38 CET 2020
ELF File Type:  executable
ELF Original Image Base:0x10000
ELF Prelinked:  false
ELF Source File [   0]: /development/abe-toolchain-build/abe.build******
ELF Source File [   1]: /development/abe-toolchain-build/abe.build/***/csu/start.o
ELF Source File [   2]: init.c
ELF Source File [   3]: /opt/linaro-armv7ahf-2015.11-gcc5.2/bin/../arm-**
ELF Source File [   4]: /opt/linaro-***-gnueabihf/libc/usr/lib/crtn.o
ELF Source File [   5]: StreamUrlsGenerator.cpp
ELF Source File [   6]: OnvifRecordingSession.cpp
ELF Source File [   7]: VideoMonitor.cpp
ELF Source File [   8]: SnapManagerService.cpp
ELF Source File [   9]: Session.cpp
ELF Source File [2880]: defs.cpp
ELF Source File [2881]: dhLBPCLDATA.c
Executable Format:      Executable and Linking Format (ELF)
Executable Location:    /root/Master/fuzzing/sonia
Executable MD5: 6b9cf840474d1413b2d94618b1ce454d
Executable SHA256:      c45d20c88a5d33f316301a409cc672856de018b3a9ee90315588b00957b57310
FSRL:   file:///root/Master/fuzzing/sonia?MD5=6b9cf840474d1413b2d94618b1ce454d
Relocatable:    false
```

A2 NAMESPACE DAHUA::DVRIP

```
146 CDVRIPConfigManager
 71 CmdsConfigUpdate
 58 CDHCommand0xF4
 41 CmdsDVRInfo
 30 CNetWorkService
 27 CPtzTransform
 22 CfgEncodeProcessor
 20 CNetDynamicManager
 18 CmdsAccount
 16 CSession
 16 CmdsF6Process
 16 CmdsCtrlProc
 15 CExtCfgInfoProcessor
 15 CDyOperation
 14 CVideoCaptureHelper
 12 CmdsUserLogin
 11 RecordProcess
 11 CmdsRecDownload
 10 CmdsDownloadFile
 10 CfgAgendaProcessor
  9 CmdsTransmitFile
  8 CmdsShareCom
  8 CMCastCaptureHelper
  8 CDvripUdpServer
  7 CLogManager
  7 CLocalAlarmSetCompact
  7 CLoadExportBlackWhite
  7 CControlDeviceCompact
  6 DVRIPThreadPool
  6 CSystemUpgradeCompact
  6 CSnapCaptureHelper
  6 CShareCommCompact
  6 CmdsSysUpdate
  6 CfgDspEncodeCapProcessor
  5 CTalkBackProcess
  5 CTalkBackCompact
  5 CQueryWorkStateCompact
  5 CQueryConfigCompact
  5 CPlaybackCompressHelper
  5 CmdsOnAudioData
  5 CmdsNetKeyBoard
  5 CFrame
  5 CDownloadFile
  5 CAutoRegisterManager
  4 DVRIPSvr
  4 CVideoCaptureAdapter
  4 CModifConfigCompact
  4 CAutoRegTask
  3 Helper
  3 CTransmitFile
  3 CSendShareCommData
  3 CQuerySysInfoCompact
  3 CNetDynamicProc
  3 CMessagePackingF8
  3 CMediaFileTransmitManager
  3 CMediaFileTransmit
  3 CMediaFileFindManager
  3 CmdsTestDevice
  3 CfgRecordProcessor
  3 CfgEventProcessor
  3 CEncrypt
  3 CCfgJsonParse_Check
  3 CBandLimitManager
  3 CAudioMonitorCompact
  2 IdPool
  2 CTimeManagerCompact
  2 CSubLinkTransmit
  2 CSubLinkTranProcess
  2 CSnapAdapter
  2 CPacking
  2 CNetAlarm
  2 CModifyChannelTitleCompact
```

```
      2 CMediaFileFinder
      2 CMCastCaptureManager
      2 CfgDevRecordGroupProcessor
      2 CF5ServiceCompact
      2 CDHCommand0x24
      2 CCfgValueBaseProcessor
      1 StrToJson(std
      1 set_tcp_keepalive_timeout(int)
      1 readremote(Dahua
      1 printfErrorInfo(int)
      1 praseKeyValue(std
      1 NotifyDownNVR(Dahua
      1 nonblock(int)
      1 NetSSLHandShakeComplete(Dahua
      1 NetSendDataToSocketBuffer(Dahua
      1 NetInt2Str(int)
      1 NetCoreAccept(Dahua
      1 LastCommandPrint()
      1 isLittleEndian()
      1 isFree(int)
      1 IPv6ConvertToIpv4(char*,char*)
      1 getPreviewChannel(std
      1 getOperator(Dahua
      1 GetNameValue(char_const*,char_const*,char*,int)
      1 getFreePort()
      1 FindKeyStr(std
      1 FilterVideoMonitor(Dahua
      1 FilterCompat(Dahua
      1 CSnapManager
      1 CScreenSplitCompact
      1 CPtzControlCompact
      1 CProtocolHead
      1 CParse
      1 COperation
      1 conn_send(Dahua
      1 CNetAlarmInputCompact
      1 CMessagePacking
      1 CLogTypeTransform
      1 CLoginInfoCompact
      1 CDVRIPMagicBox
      1 CDVRIP
      1 CDPSValueName
      1 CDPStrParse
      1 CDHCommand0xF5
      1 CDHCommand0xD0
      1 CDHCommand0xC6
      1 CDHCommand0xA8
      1 CDHCommand0xa1
      1 CDHCommand0x61
      1 CDHCommand0x23
      1 CCtrlIFrameCompact
      1 CCaptureHelper
      1 CAutoRegisterCompact
      1 CAuthorityManager
      1 CAlgorithm
      1 CAlarmState
```

A3 WIRESHARK DISSECTOR: DVRIP LUA SCRIPT

```
-- Definition of the overall protocol name
dahua_proto = Proto("dahua","Dahua DVRIP Protocol")

-- Protocol tree fields shown in Wireshark
DVRIP_b1_message_type =          ProtoField.uint16("dvrip.type",         "b1 - Message Type        ",
base.HEX)
DVRIP_b2_payload_length =        ProtoField.uint16("dvrip.length1",      "b2 - Payload Length      ",
base.HEX)
DVRIP_b3_sequence_id =           ProtoField.uint16("dvrip.sequence",     "b3 - Sequence ID         ",
base.HEX)
DVRIP_b4_unknown_1 =             ProtoField.uint16("dvrip.unknown1",     "b4 - Unknown             ",
base.HEX)
DVRIP_b5_payload_length_sid =    ProtoField.uint16("dvrip.length2",      "b5 - Payload Length / SID",
base.HEX)
DVRIP_b6_unknown_2 =             ProtoField.uint16("dvrip.unknown2",     "b6 - Unknwon             ",
base.HEX)
DVRIP_b7_sessionID =             ProtoField.uint16("dvrip.sessionID",    "b7 - SessionID           ",
base.HEX)
DVRIP_b8_zero_type =             ProtoField.uint16("dvrip.zero_type",    "b8 - Zero / LoginType    ",
base.HEX)
DVRIP_b9_payload =               ProtoField.uint16("dvrip.payload",      "b9 - Payload Body        ",
base.json_string_f)
DVRIP_b9_payload_JSON_RAW =      ProtoField.string("dahua.data",         "Raw Message")

-- Adding the previous diefined protocol fields to the protocol
dahua_proto.fields = {
    DVRIP_b1_message_type,
    DVRIP_b2_payload_length,
    DVRIP_b3_sequence_id,
    DVRIP_b4_unknown_1,
    DVRIP_b5_payload_length_sid,
    DVRIP_b6_unknown_2,
    DVRIP_b7_sessionID,
    DVRIP_b8_zero_type,
    DVRIP_b9_payload,
    DVRIP_b9_payload_JSON_RAW,
}

-- Loading JSON API
local json = Dissector.get("json")

-- Main definition of the protocl dissector
function dahua_proto.dissector(buffer,pinfo,tree)
    -- checking for zero length
    length = buffer:len()
        if length == 0 then return end

        pinfo.cols.protocol = dahua_proto.name

    -- naming the protocol in menue
    local subtree = tree:add(dahua_proto, buffer(), "Dahua DVRIP Protocol")

    -- adding the protocol fields
    local b1_tree = subtree:add(DVRIP_b1_message_type, buffer(0,4))
    local b2_tree = subtree:add(DVRIP_b2_payload_length, buffer(4,4))
    local b3_tree = subtree:add(DVRIP_b3_sequence_id, buffer(8,4))
    local b4_tree = subtree:add(DVRIP_b4_unknown_1, buffer(12,4))
    local b5_tree = subtree:add(DVRIP_b5_payload_length_sid, buffer(16,4))
    local b6_tree = subtree:add(DVRIP_b6_unknown_2, buffer(20,4))
    local b7_tree = subtree:add(DVRIP_b7_sessionID, buffer(24,4))
    local b8_tree = subtree:add(DVRIP_b8_zero_type, buffer(28,4))

    -- search for JSON in the payload field
    if buffer:len() >32 then
        -- detect and load JSON values
        payload_tvbrange = buffer.range

        if buffer(32,1):string() == "{" then
            -- loading buffer
            local tvb_uncompress = buffer(32,buffer:len()-32)

            -- raw text
```

```
            local b9_tmp = subtree:add(DVRIP_b9_payload_JSON_RAW, tvb_uncompress)

            -- as JSON structure
            local test = json:call(tvb_uncompress:tvb(), pinfo, subtree)

        end

    end

end

-- assigning protocol to port
tcp_table = DissectorTable.get("tcp.port")
tcp_table:add(37777,dahua_proto)
```

A4 SDK DEMO APPLICATIONS

Demo applications inside the SDK:

- AccessControl.exe
- AlarmDemo.exe
- AlarmDevice.exe
- Alarm.exe
- CapturePicture.exe
- DeviceControlAndTimeSynchronization.exe
- DevInit.exe (used for the DHIP fuzzing case)
- ImageTest.exe
- IntelligentDevice.exe
- IntelligentTrafficDemo.exe
- IntelligentTraffic.exe
- MonitorWall.exe
- NetSDKDemo.exe
- PlayBackAndDownloadDemo.exe
- PlayBack.exe
- RealPlayAndPTZControl.exe
- RealPlayAndPTZDemo.exe
- ShowDemo.exe
- TalkDemo.exe
- Talk.exe
- TestOSD.exe
- ThermalCamera.exe

A5 CUSTOM SDK APPLICATION

```
//============================================================================
// Name        : testcase 001
// Author      : Thomas Vogt
// Version     : 2020-02-17
// Description : Simple Login and Logout Test
//============================================================================

#include <iostream>
#include <unistd.h> // for sleep
#include <string>
#include "dhnetsdk.h"
using namespace std;

LLONG m_lLoginHandle;

void CALLBACK DisConnectFunc(LLONG lLoginID, char *pchDVRIP, LONG nDVRPort, LDWORD dwUser)
{
    cout << "Callback DisConnect" << endl;
    return;
}

void init()
{
    cout << "SDK Initialization" << endl;
    bool success = CLIENT_Init(DisConnectFunc, (LDWORD) 0);
    if (success)
    {
        cout << "init ok" << endl;
    }
    else
    {
        cout << "init failed" << endl;
    }
}


bool login(string destip, long port, string username, string password)
{
    cout << "login" << endl;
    int error = 0;
    NET_DEVICEINFO deviceInfo = {0};


    m_lLoginHandle   =   CLIENT_Login(destip.c_str(),   port,   username.c_str(),   password.c_str(),
&deviceInfo, &error);

    if(m_lLoginHandle == 0)
    {
        if(error != 255)
        {
            cout << "login failed with code " << error << endl;
        }
        else
        {
            cout << "login also failed" << endl;
        }
        return false;
    }
    else
    {
        cout << "login ok" << endl;
        return true;
    }

}

void logout()
{
    cout << "logout" << endl;
    bool success = CLIENT_Logout(m_lLoginHandle);
    if(success)
    {
```

```
        cout << "logout ok" << endl;
    }
    else
    {
        cout << "logout failed" << endl;
    }
}


int main(int argc, char* argv[]) {
        cout << "Dahua Login Test" << endl;
    if (argc != 5)
    {
        cout << "Usage: xxxxx <host ip> <port> <username> <password>" << endl;
        return 1;
    }

    init();
    cout << "initialized" << endl;
    sleep(2);
    bool log_ok = login(argv[1], std::stol(argv[2]), argv[3], argv[4]);
    if (!log_ok)
    {
        return 2;
    }
    cout << "Login successfull" << endl;

    cout << "Logout in 2 seconds" << endl;


    //triggerAlarm(alarmIndex, 1);
    sleep(2);
    //triggerAlarm(alarmIndex, 0);

    logout();
        return 0;
}
```

A6 CVE-2020-9502: SESSION HIJACKING SECURITY ADVISORY

The following security advisory was sent to Dahua Product Security Incident Response Team. The information and data were sent via encrypted mail.

```
Product :               Dahua IPC-HFW1431S
CVE:                    CVE-2020-9502
CAPEC:          CAPEC-593: Session Hijacking
Severity:               High
Vendor Hompage: https://www.dahuasecurity.com/
Found:                  01/31/2020
Researcher:             Thomas Vogt

CVSS Base Score
CVSSv3 Overall Score:   8.8 (High)
CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H


Vulnerability Description

The Dahua binary P2P protocol on default port 37777 contains a session hijacking vulnerability. This
protocol is used to configure, use and maintain devices in the categories camera. The adversary is
able to steal an active session and use it to gain unauthorised access to the target application.
Impact Description IPC-HFW1431S

A successful session hijacking can result in a full compromise of the target device. This includes
all actions that can be performed with the official vendor configuration tool ConfigTool, which
operates over port 37777.


Impact Description Worldwide

The process and behaviour how the Dahua P2P Protocol handle and creates SessionID's for a new active
session was observed at the usage of the ConfigTool and the usage of the official Dahua SDK. With the
fact that the SDK provides no special API or methods for different device types, brings the assumption
that all network devices from Dahua could be affected from this protocol vulnerability. This applies
especially for categories Camera, Intelligent Analysis Server, Intelligent Building, Intelligent
Traffic, Storage and maybe others, which are named in the SDK documentation. This vulnerability could
affect over 1. Million devices available currently on the Internet.


Proof of Concept and Technical Description (for internal usage only)

The session hijacking is possible due to a very poorly session entropy. After a successful login on
the target application, the application is sending an active session identifier for the client, named
SessionID in this description. This SessionID is used in all later communication, either as binary
value or as integer (little-endian) in JSON requests.

Example of SessionID in the binary Header:
'83ffff7f'
'82ffff7f'
'81ffff7f'
'80ffff7f'
'7fffff7f'

Example of SessionID in a JSON request:
{"method": "global.keepAlive", "magic": "0x1234", "params": {"timeout": 30, "active": true}, "id": 5,
"session": 2147483523}

The red Integer 2147483523 is the binary value of '83ffff7f' (UINT32 – Little Endian(DCBA))

Commands from over the P2P Protocol on port 37777 with the ConfigTool are issued either directly with
the Binary Header or over the P2P service API, accessed via JSON requests. A service API call looks
like the following code:

{"method": "global.keepAlive", "magic": "0x1234", "params": {"timeout": 30, "active": true}, "id": 5,
"session": 2147483523}

All services already more known accessible over HTTP are available over the service API. This include
services mentioned at:
```

https://s3.amazonaws.com/amcrest-files/AMCREST_HTTP_API_SDK_V2.10.pdf
https://s3.amazonaws.com/amcrest-files/Amcrest+HTTP+API+3.2017.pdf

Services and Methods can also be retrieved with the following request:
{"method": "system.listService", "session": 2147483640, "params": null, "id": 3}

Response:
{"id":3,"params":{"service":["system","AC","AccessCard","AccessFace","AccessFingerprint","AccessUser","Authentication2th","BlueToothDeviceManager","CertManager","CloudUpgrader","CoaxialAlarmManager","CoaxialControlIO","CoaxialEQ","CoaxialPoC","CoaxialSensorManager","CoaxialWhiteLightSpeaker","DM","DataFlux","DataOutputVkManager","DockUser","FileManager","FtpTest","GpsControl","Gyro","HeatMap","IntervideoManager","LensManager","MediaEncrypt","Nat","OSDManager","PasswdFind","PirAlarm","PositionManager","RPCTest","RecordFinder","SDEncrypt","Security","StreamUrlService","UPnPPortmap","VehicleKeyConfig","VehicleSensor","VideoInAnalyse","VideoProcessManager","WlanAutoTest","alarm","bus","configManager","console","ddnsClient","devAudioDetect","devAudioEncode","devAudioInput","devAudioOutput","devStorage","devVideoAnalyse","devVideoDetect","devVideoEncode","devVideoInput","devVideoOutput","deviceDiscovery","encode","eventManager","faceBoard","global","intelli","locales","log","magicBox","mediaFile","mediaFileFind","mobile","netApp","printLog","ptz","rainBrush","recordManager","snapManager","speak","storage","upgrader","userManager","videoStatServer","workDirectory","workGroup"]},"result":true,"session":2147483640}

Authentication and Security of the Session Identifier; a successful login looks like in the following picture:

```
12  [[[ client <> server login 4 way Handshake ]]]
13  Message-Type | Payload-Length | ID  msg-num| unknown-1  | SessionID  | unknown-2  | SessionID  | unknown-3  | Payload
14  ------------ | -------------- | ---------- | ---------- | ---------- | ---------- | ---------- | ---------- | -------------------------
15  'a0050060'   | '00000000'     | 'c4a3af48' | '9956b6b4' | 'fa269dec' | '29d84afb' | '05020001' | '0000a1aa' | ''
16  'b0000078'   | '45000000'     | '010e0100' | '00000000' | '00000000' | '01000000' | '0600f900' | '00000002' | 'Realm:Login to 421592117a
17  'a0050060'   | '47000000'     | '00000000' | '00000000' | '00000000' | '00000000' | '05020008' | '0000a1aa' | 'admin&&61976C2F7CD6D761C1
18  'b0000078'   | '00000000'     | '00080100' | '33000000' | 'f9ffff7f' | '01000000' | '0600f900' | '00050002' | ''
```

The Dahua P2P is above a TCP connection, so the shown bytes are send inside a TCP connection

Line 15: Bytes send from Client to Initiate a Session
Line 16: Bytes from the target device, contains a random value for Hash calculation
Line 17: Bytes sent from Client, contains Hash with password and random value
Line 18: Bytes from the target device, contains SessionID after valid Hash was sent in msg3

For a successful session hijacking, an adversary must only know this SessionID value. No password is needed. No username is needed. This SessionID is only changing 1 byte after each successful login on the target device. With the fact that the SessionID has almost no randomness and entropy, it is very likely to gain unauthorised access with this attack. An additional behaviour was observed, that the SessionID was always starting with the same value when the device was rebooted. Predicting the SessionID or brute-forcing the SessionID is thus extremely easy.

Affected Products
The vulnerability was identified on the model IPC-HFW1431S, System Version V2.800.0000004.0.R, Build Date: 2019-01-22, WEB Version V3.2.1.688161 and Security Baseline Version V1.4.

Please see Impact Description Worldwide. This could be a Protocol vulnerability itself and affect all Products which used this protocol, white-labels and other vendors from Dahua included.

Timeline
01/31/2020: Vulnerability identified
01/31/2020: Dahua PSIRT informed via encrypted mail
02/11/2020: Dahua requests additional details and exploit
02/12/2020: Provided Dahua a working exploit as PoC and video PoC
02/12/2020: Dahua confirms retrieved email
02/15/2020: Technical analysis still in queue
02/25/2020: Dahua confirms vulnerability and arrange repair plan before May 12
03/06/2020: CVE-2020-9502 assigend
05/12/2020: Dahua released official SA and updated firmwares

Figure 70: Security advisory for CVE-2020-9502

A7 DEBUG VERSION OF MAIN SONIA BINARY

The following information and data were sent to Dahua to report a potential security vulnerability. The data was sent via encrypted mail.

```
Short Description + Impact: Unprotected Source-Code, this can reveal secrets, algorithms from your AI
parts, all mechanisms inside the Code. That means with more detail:

    • full recovery of over 2800 ELF Source Files (e.g. cpp and c files, e.g. DVRIPConfigManager.cpp)
    • C(PP) code from all functions (over 70.000)

Due to the fact that it seems that some (a lot) firmware also when downloaded from your main site has
the debug version, the debug version of the sonia binary should be wide spread.

Example Firmware containing Sonia Debug Version (binary has 200MB):
https://www.dahuasecurity.com/support/downloadCenter/download-search?keyword=HX2X
Chosen Language (influences search results): International - English
```

Figure 71: Security advisory for debug version of main Sonia binary

A8 HELPDESK FIRMWARE ACCESS OF ALL DAHUA PRODUCTS

The following information and data were sent to Dahua to report a potential security vulnerability. The data was sent via encrypted mail.

```
Short Description: All firmware (sorted after firmware baseline etc) can be downloaded over an helpdesk
account, accessible worldwide.

Depending on your internal argumentation and risk assessment vulnerability #2 could be a vulnerability
with high, low or none risk.

    • https://dahua.support/en/support/solutions
    • https://dahua.support/en/support/solutions/articles/75000018589-firmware
    • https://dahua.support/en/support/solutions/articles/75000019881-firmware
    • https://dahua.support/en/support/solutions/articles/75000018598-nvr-firmware
    • https://dahua.support/en/support/solutions/articles/75000018607-vdp-firmware
    • https://dahua.support/en/support/solutions/articles/75000018608-asc-firmware
    • https://dahua.support/en/support/solutions/articles/75000018634-hdcvi-record-frimware

Login link:
In Europe:  https://share.dahuasecurity.com
In China: https://share.dahuasecurity.com:8080/
Username:dahuah********** // cleartext username is hidden due to security reasons
Password:dahua2********** // cleartext password is hidden due to security reasons

Potential Risk: Unauthorised access of firmware and potential upload of firmware files (upload button
showed). If the upload is working that could be critical due to the fact that firmware with malware
could be uploaded and used then by other helpdesk users.

Recommendation: Check if the Share (SharePoint?) Access is allowed to authorised to be public. If yes,
please give me feedback, because I know there are a lot of other researcher / people which are very
interested in that firmware.
```

Figure 72: Security advisory for helpdesk access of all Dahua products

A9 FULL LOGENGINE OUTPUT OF ALL CLIENT SIMULATION SCRIPT

```
12:55:13|[RemoteService] info tid:3268 a client from ip: ::ffff:192.168.178.20, port 18051, socket
38 scope_id 0
12:55:13|[RemoteService] debug tid:3268 Set tcp tos 0 in accept.
12:55:13|[RemoteService] debug tid:3268 Set udp 67 tos 0 in accept.
12:55:13|[Manager] info tid:3268 CUser::CUser(0x0x442adb08)>>>>>
12:55:13|[RemoteService] trace tid:3268 IPv6ConvertToIpv4 succeed>IPv6:::ffff:192.168.178.20,
IPv4:192.168.178.20
12:55:13|[RemoteService] trace tid:3268 The New Conn IP(192.168.178.20)...
12:55:13|[RemoteService] trace tid:3268 CNetWorkService::isValid not yet login !
12:55:13|[RemoteService] trace tid:3268  =======>this device Type: BURGcamBULLET, device Class: IPC
12:55:13|[RemoteService] info tid:3268 CmdsUserLogin::OnLogin() remote sock:38, session:280,
loginConnFlag:0, iSubConnFlag:0
12:55:13|[] debug tid:3268 tracepoint: Src/DVRIP/Cmds_UserLogin.cpp, 255.
12:55:13|[RemoteService] info tid:3268 m_DVRIP.dvrip.dvrip_p[16] = 0
12:55:13|[RemoteService] error tid:3268 m_pDVRIPHead->dvrip.dvrip_p[19]:1
12:55:13|[RemoteService] trace tid:3268 [DVRIP] DES len is 128 ,strUName.size:8,strUPwd.size:8
12:55:13|[RemoteService] trace tid:3268 --------DES_LOGIN:loginState: getRandomInfo---------------
12:55:13|[RemoteService] info tid:3268 login type:DES_LOGIN, userName:admin, password:*******
12:55:13|[RemoteService] trace tid:3268 net  client verison 6 ,iClientFlag = 1
12:55:13|[RemoteService] trace tid:3268 iLoginCount:0, iMaxConn:20
12:55:13|[RemoteService] trace tid:3268 >>>>>>>>>loginType:DES_LOGIN, loginState:getRandomInfo
12:55:13|[RemoteService] trace tid:3268 DVRIP getRandominfo
12:55:13|[RemoteService] trace tid:3268 Geting RandomInfo frome RPCServer!
12:55:13|[RemoteService] trace tid:3239 CGlobal::login
12:55:13|[RemoteService] trace tid:3239 CGlobal::login first time
12:55:13|[RemoteService] trace tid:3239 login num:0 MaxConn:10
12:55:13|[RemoteService] info tid:3239 CLogin::firstLogin dhipUser->random:2010427978, dhipUser-
>session:355583564!
12:55:13|[RemoteService] trace tid:3268 CDVRIPRPCClient::call() buffer:
{ "error" : { "code" : 401, "message" : "unauthorized" }, "id" : 36, "params" : { "encryption" :
"Default", "random" : "612295939", "realm" : "Login to 803906081050579" }, "result" : false,
"session" : 355583564 }
12:55:13|[] debug tid:3268 tracepoint: Src/DVRIP/DVRIPRPCClient.cpp, 1453.
12:55:13|[RemoteService] trace tid:3268 strloginType is DES_LOGIN
12:55:13|[RemoteService] trace tid:3268 CNetWorkService::isValid not yet login !
12:55:13|[RemoteService] trace tid:3268  =======>this device Type: BURGcamBULLET, device Class: IPC
12:55:13|[RemoteService] info tid:3268 CmdsUserLogin::OnLogin() remote sock:38, session:355583564,
loginConnFlag:2, iSubConnFlag:0
12:55:13|[] debug tid:3268 tracepoint: Src/DVRIP/Cmds_UserLogin.cpp, 255.
12:55:13|[RemoteService] info tid:3268 m_DVRIP.dvrip.dvrip_p[16] = 0
12:55:13|[RemoteService] error tid:3268 m_pDVRIPHead->dvrip.dvrip_p[19]:8
12:55:13|[RemoteService] trace tid:3268 [DVRIP]--UserLogin extra data !!!
12:55:13|[] debug tid:3268 tracepoint: Src/DVRIP/Cmds_UserLogin.cpp, 333.
12:55:13|[RemoteService] info tid:3268 DVRIP:strUName=admin
12:55:13|[RemoteService] trace tid:3268 --------DES_LOGIN:loginState: Login---------------
12:55:13|[RemoteService] info tid:3268 login type:DES_LOGIN, userName:admin, password:*******
12:55:13|[RemoteService] trace tid:3268 net  client verison 6 ,iClientFlag = 1
12:55:13|[RemoteService] trace tid:3268 iLoginCount:0, iMaxConn:20
12:55:13|[RemoteService] trace tid:3268 >>>>>>>>>loginType:DES_LOGIN, loginState:Login
12:55:13|[RemoteService] trace tid:3240 CGlobal::login
12:55:13|[RemoteService] trace tid:3240 CGlobal::login second time
12:55:13|[Manager] info tid:3240 CUser::CUser(0x0x333cf78)>>>>>
12:55:13|[Manager] info tid:3240 CLocalClient::CLocalClient(0x0x30b5168)>>>>>>
12:55:13|[Manager] info tid:3240 si.loginType=0, si.clientAddress=192.168.178.20,
si.clientType=Local si.authorityInfo= si.authorityType= si.passwordType=Plain
12:55:13|[Manager] trace tid:3240 IsIPContainedIn  [127.0.0.1-127.0.0.1], [192.168.178.20]
12:55:13|[Manager] debug tid:3240 CUserManager::checkPassword() user 'admin' use mutable super
password failed, pwd:6C51FE54771B7C3DB45EE34D4F1EAF47.
12:55:13|[Manager] debug tid:3240 CLocalClient::login() successful! username = admin
12:55:13|[Manager] info tid:3240 CUserManager:: AddActiveUser:admin 0x333cf78, id:15
12:55:13|[RemoteService] trace tid:3240 login successful
12:55:13|[RemoteService] trace tid:3268 CDVRIPRPCClient::call() buffer:
{ "id" : 37, "params" : null, "result" : true, "session" : 355583564 }
12:55:13|[RemoteService] trace tid:3241 keepalive successful session:355583564
12:55:13|[Manager] info tid:3268 Login type=DVRIP, typeUserManager=DES_LOGIN
12:55:13|[Manager] info tid:3268 CLocalClient::CLocalClient(0x0x436bf0e0)>>>>>>
12:55:13|[Manager] info tid:3268 si.loginType=0, si.clientAddress=192.168.178.20,
si.clientType=DVRIP si.authorityInfo= si.authorityType= si.passwordType=Plain
12:55:13|[Manager] trace tid:3268 IsIPContainedIn  [127.0.0.1-127.0.0.1], [192.168.178.20]
12:55:13|[Manager] debug tid:3268 CUserManager::checkPassword() user 'admin' use mutable super
password failed, pwd:6C51FE54771B7C3DB45EE34D4F1EAF47D40D637CF5F7BF70CF21ED1E8949970C.
```

```
12:55:13|[Manager] debug tid:3268 CLocalClient::login() successful! username = admin
12:55:13|[Manager] info tid:3268 CUserManager:: AddActiveUser:admin 0x442adb08, id:16
12:55:13|[RemoteService] trace tid:3268 Login(admin, *******)
12:55:13|[RemoteService] trace tid:3268 Login SendBackMessage:0 session:1
12:55:13|[libInfra] debug tid:3318 ThreadBody Enter name = DVRIPRPCClient, id = 3318, prior = N10,
stack = 0x45543e2c
12:55:30|[RemoteService] warn tid:3268 CmdsF6Process offlineManagerPtr init failed
12:55:30|[Manager] info tid:3268 setCurrentUserID:2147483631 <==> 16
12:55:30|[RemoteService] trace tid:3237 CSystemOperatorService::getDeviceType
12:55:30|[RemoteService] trace tid:3237 CSystemOperatorService::getDeviceType successful
12:55:30|[RemoteService] trace tid:3242 CSystemOperatorService::getDeviceType
12:55:30|[RemoteService] trace tid:3242 CSystemOperatorService::getDeviceType successful
12:55:30|[RemoteService] trace tid:3241 CSystemOperatorService::getSerialNo
12:55:30|[RemoteService] trace tid:3241 CSystemOperatorService::getSerialNo successful
12:55:30|[RemoteService] trace tid:3238 CSystemOperatorService::getVendor
12:55:30|[Manager] trace tid:3238 get Vendor: BurgWaechter
12:55:30|[RemoteService] trace tid:3238 CSystemOperatorService::getVendor successful
12:55:32|[RemoteService] error tid:3239 UtilityRPC::jsonToUInt, can't change json to uint
12:55:32|[Manager] info tid:3240 User Name        Login As         Remote Address:
12:55:32|[Manager] info tid:3240 _____________________________________________
12:55:32|[Manager] info tid:3240 admin           Local            192.168.178.20
12:55:32|[Manager] info tid:3240 admin           DVRIP            192.168.178.20
readremote: Connection reset by peer
12:55:34|[RemoteService] trace tid:3268 DVRIP readremote[socket is 38]
12:55:34|[] debug tid:3268 tracepoint: Src/DVRIP/NetCore.cpp, 2062.
12:55:34|[Manager] warn tid:3268 CCommonConfigManager::getConfig EmergencyRecordForPull is
Json::nullValue!
12:55:34|[RemoteService] trace tid:3268
CmdsUserLogin::NotifyDownNVR():getconfig:EmergencyRecordForPull faied
1587041734 DVRIP [disconnecting 0x4366a240]
12:55:34|[RemoteService] trace tid:3268 CaptureHelper::stopSnapCapture>>>>>channel 0 has stoped
snapCapture
12:55:34|[NetFramework] warn tid:3240 Src/MediaStreamSender.cpp:690 Clear
CMediaBuffer::Clear,m_frame_header:(nil)
12:55:34|[RemoteService] trace tid:3268 CDVRIPRPCClient::logout! session : 355583564
12:55:34|[RemoteService] trace tid:3240 CGlobal::logout
12:55:34|[Manager] info tid:3240 client logout type:Local, logout
12:55:34|[Manager] info tid:3240 CUserManager:: RemoveActiveUser:admin
12:55:34|[RemoteService] trace tid:3240 CDevVideoEncodeService::stopDevVideoEncodeService
12:55:34|[RemoteService] trace tid:3240 CMeidaFileService::closeMediaFileService
12:55:34|[RemoteService] trace tid:3240 CDevAudioEncodeService::stopDevAudioEncodeService
12:55:34|[RemoteService] info tid:3240 CLogin::deleteUser random:2010427978, session:355583564,
12:55:34|[Manager] info tid:3240 CUser::~CUser(0x0x333cf78)>>>>>
12:55:34|[Manager] info tid:3240 CLocalClient::~CLocalClient(0x0x30b5168)>>>>>>
12:55:34|[RemoteService] trace tid:3240 logout successfule
12:55:37|[libInfra] debug tid:3318 ThreadBody leave name = DVRIPRPCClient, id = 3318
12:55:37|[RemoteService] trace tid:3268 CDVRIPRPCClient::closeSocket
12:55:37|[RemoteService] trace tid:3268 CDVRIPRPCClient::logout
....session:355583564....clients:0.....
12:55:37|[Manager] info tid:3268 client logout type:DVRIP, logout
12:55:37|[Manager] info tid:3268 CUserManager:: RemoveActiveUser:admin
12:55:37|[Manager] info tid:3268 CUser::~CUser(0x0x442adb08)>>>>>
12:55:37|[Manager] info tid:3268 CLocalClient::~CLocalClient(0x0x436bf0e0)>>>>>>
12:55:37|[RemoteService] warn tid:3237 Src/RPCServer/rpc_server/SubSocketHandler.cpp:132
handle_input error this:0x30b7f28
12:55:37|[RemoteService] trace tid:3241 RPCS 2020-04-16 12:55:37 !!! close CStreamSenderSource
12:55:37|[RemoteService] info tid:3241 CMediaStreamServer::handle_message:fromId:-106491 ,this-
>GetId:-102143, type:12292
12:55:37|[RemoteService] trace tid:3241 RPCS 2020-04-16 12:55:37 !!! close CMediaStreamSource
```