University of Tartu
Institute of Computer Science
Computer Science Curriculum

Rico-Andreas Lepp

Enhancing Single Packet Authorization with Multi-Factor Authentication

Bachelor's Thesis (9 ECTS)

Supervisor:
Tarmo Oja, MSc

Tartu 2025

Enhancing Single Packet Authorization with Multi-Factor Authentication

Abstract:

Single Packet Authorization (SPA) is a network security technique for adding an additional layer of authentication and concealing services from adversaries while still allowing access for authorized users. This shifts the attack surface from the services to the SPA implementation itself, making its security crucial. Publicly available implementations of SPA use a shared static secret, leaving a single point of failure in case the secret is compromised. One possible solution to this issue is the usage of Multi-Factor Authentication (MFA) with the Time-based One-Time Password (TOTP) system, which uses short-lived codes generated on an external device. As a result of this thesis, the TOTP system was integrated with a specific implementation of SPA known as *fwknop*.

Keywords: Single Packet Authorization (SPA), MFA, TOTP, *fwknop*

CERCS: P175 Informatics, systems theory

Ainu-pakett autentimise täiendamine mitmikautentimisega

Lühikokkuvõte:

Ainu-pakett autentimine (SPA) on võrguturbe tehnoloogia, mis võimaldab peita teenuseid ründajate eest, lisades uue autentimiskihi, kust pääseb läbi ainult lisaautentimise läbinud kasutaja. Nii nihkub ründepind teenustelt SPA enda peale, mistõttu muutub ka selle turvalisus oluliseks. Olemasolevad SPA implementatsioonid kasutavad ühissaladust, mistõttu on aga kogu süsteemil keskne nõrk lüli. Üks võimalik leevendus sellele on rakendada mitmikautentimist (MFA) kasutades ajapõhise parooli (TOTP) süsteemi ning hoides algsaladust lisaseadmes. Lõputöö tulemusena lisati ühele SPA tööriistale nimega *fwknop* mitmikautentimise võimalus TOTP süsteemi abil.

Võtmesõnad: ainu-pakett autentimine, mitmikautentimine, TOTP, *fwknop*

CERCS: P175 Informaatika, süsteemiteooria

Contents

Abbreviations

AES Advanced Encryption Standard. 3, 26

API Application Programming Interface. 3, 12, 27

CVE Common Vulnerabilities and Exposures. 3, 10

DoS Denial of Service. 3, 7, 10, 27

GPG The GNU Privacy Guard. 3, 13, 18, 26, 27

HMAC Hash-based Message Authentication Code. 3, 20, 25, 26, 30

HOTP HMAC-based One-Time Password. 3, 20, 21, 23

HTTP Hypertext Transfer Protocol. 3, 9

IANA Internet Assigned Numbers Authority. 3, 9

ICMP Internet Control Message Protocol. 3, 14

IoT Internet of Things. 3, 7, 19, 25

MFA Multi-Factor Authentication. 2, 3, 7, 18, 19, 23

MitM Man-in-the-Middle. 3, 19

NAT Network Address Translation. 3, 10, 17, 25

OTP One-Time Password. 3, 18, 20, 23

PK Port-Knocking. 3, 10

PKI Public-Key Infrastructure. 3, 18

PoC Proof-of-Concept. 3, 10

SPA Single Packet Authorization. 2, 3, 7, 11, 12, 15, 17–19, 23–25, 27, 31–34, 37, 38

SSH Secure Shell. 3, 9

TCP Transmission Control Protocol. 3, 10, 14

TLS Transport Layer Security. 3, 14

TOTP Time-based One-Time Password. 2, 3, 7, 8, 21–24, 29, 30, 32–34, 37, 38, 41

UDP User Datagram Protocol. 3, 14

VPN Virtual Private Network. 3, 10

1. Introduction

Internet of Things (IoT) devices are increasing in popularity due to their value as small but efficient data collectors. These devices often have a specific purpose and thus carry very limited functionality and computation power. While the management of these systems needs to be simple and efficient, security is also a crucial point to keep in mind. Cyber attacks are a growing threat on all fronts against organizations of all sizes. It is of the utmost necessity to protect these devices in order to keep the data flow and business processes running.

Even though remote access is a comfortable way for managing such devices, exposing a service to the public through an open port allows anyone to scan it, which can be used by malicious actors to retrieve information about the underlying software. This can be used in the reconnaissance phase of a cyber attack, where the adversaries try to gather as much information as possible about the target system. After the software has been identified, it may be possible to conduct credential brute-force, Denial of Service (DoS), or similar attacks. In case any publicly available exploits are not discovered, specific software versions can still make the system a target for zero-day attacks.

One possible way of restricting enumeration while still allowing remote access, is the use of Single Packet Authorization (SPA), which allows for concealing the service port with the firewall by default. Only after the user has sent a specifically crafted network packet that carries cryptographically encrypted and signed authentication information, does the SPA process allow access to the underlying service behind the firewall. The main idea of the technique is to add an additional layer of authentication; it is not a replacement for the service authentication by any means. There are numerous publicly available implementations of SPA across different platforms and programming languages; however, a common issue with all of them is the usage of a shared static secret.

The goal of this thesis is to explain the necessity of SPA, delve into the issues related to using static secrets with SPA, and propose one possible solution to this issue using Multi-Factor Authentication (MFA) with the Time-based One-Time Password (TOTP) system. The practical part of this thesis consists of integrating a TOTP system with one concrete implementation of SPA known as FireWall KNock OPerator (*fwknop*).

The thesis is organized as follows:

- Chapter 1: Introduces the topics and structure of the thesis;
- Chapter 2: Explains background information, and provides a detailed description regarding the technical aspects of SPA;
- Chapter 3: Starts by analyzing the overall security model of SPA, followed by an overview of related work and TOTP theory. Finally, examines the details of the proposed solution;
- Chapter 4: Provides the technical details and validation for the TOTP integration with *fwknop*;
- Chapter 5: Conclusions of the end results are drawn in the last section.

References to all the code written in the practical part can be found in the Appendix section.

2. Background

Networked devices utilize ports to establish reliable connections between each other. While the Internet Assigned Numbers Authority (IANA) maintains a standard for assigning ports to specific uses[1], in reality, services can use arbitrary ports. This raises the need for the server to communicate to the client what type of software is running on the open port. Depending on the service, the server either responds directly with the software and version number after a successful TCP connection, or requires the client to first send some initiation data. An example of the first case is the Secure Shell (SSH) protocol, while the latter can be seen in the Hypertext Transfer Protocol (HTTP) as shown in Figure 1.

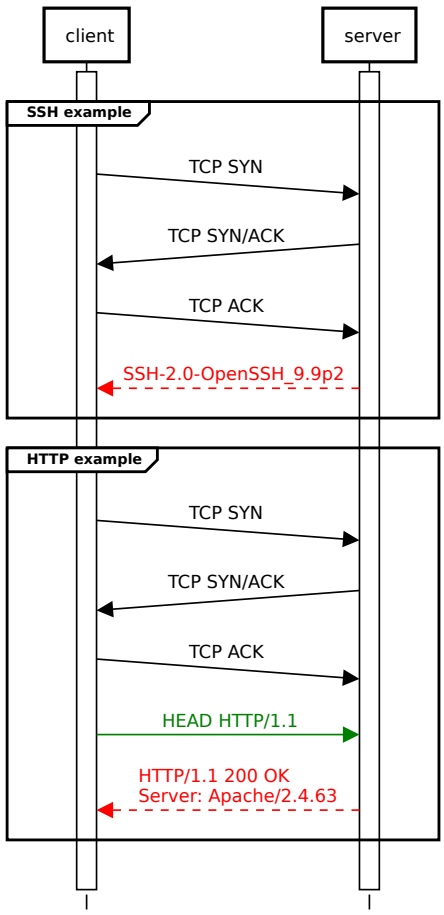

Figure 1. Service identification example.

Banner grabbing is a well-known technique used by adversaries that abuse this type of behavior in order to determine the type and version of the running software. Depending

[1]https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

on the identified service, it may be possible to conduct further attacks like credential brute-forcing or spraying, Denial of Service (DoS), or abusing direct unauthorized access if the service lacks any form of authentication. Furthermore, in case the discovered software is part of a legacy system, it may have one or multiple publicly disclosed vulnerabilities mapped in the Common Vulnerabilities and Exposures (CVE)[2] catalog. Stumbling upon a legacy service at the right time is something that even a script kiddie is able to exploit nowadays, due to vulnerability scanners like nmap[3] and Nessus[4], which leverage the banner grabbing technique and map the detected software versions to applicable CVEs. Exploitation may be carried out using penetration testing frameworks like Metasploit[5] or searching for Proof-of-Concept (PoC) scripts from specifically designed vulnerability databases.

Protecting services on the network level has had tremendous improvements over the recent decades with improvements to the network stack itself (such as SSL/TLS, etc.) and the rise of network security concepts (such as firewalls, the DMZ architecture, network segmentation, VPN, etc.) [1]. These solutions are a great way to provide protection and concealment for a large network, but setting it all up can be a tedious process and a huge overkill for use with a single device. Following all the best security practices is undoubtedly necessary, but creating a completely isolated network segment for a temporary IoT system is not worth the trouble in most cases, due to the cost of implementation being higher than the outcome value. Not to mention that implementing some of these concepts may require changes to the surrounding infrastructure due to Network Address Translation (NAT). In case the complexities of securing the system outweigh the benefits, the device may be left as a low-hanging fruit for threat actors looking to gain an initial foothold in the network.

This gave rise to the need for a simpler way to provide network-level protection against adversaries while still allowing comfortable remote access over a network connection. Port-Knocking (PK) is the first known concept that attempted to deal with this issue [2]. It essentially relies on the simple idea of sending a sequence of partial Transmission

[2]https://cve.mitre.org/
[3]https://nmap.org/nsedoc/scripts/banner.html
[4]https://www.tenable.com/products/nessus
[5]https://www.metasploit.com/

Control Protocol (TCP) connections in a predefined order, which is known to the server. The inherent flaw with this technique is that anyone eavesdropping on the network traffic would be able to replay the same sequence and gain access to the service.

A more advanced concept to provide authorized personnel temporary access over an untrusted network without exposing the service to malicious actors, uses a fundamentally different approach and is known as Single Packet Authorization (SPA).

2.1 Single Packet Authorization (SPA)

As the name suggests, Single Packet Authorization (SPA) transmits merely a single network packet from the client to the server to deliver all the necessary information for successful user authentication. In the default state, the firewall of the machine should be set to drop attempted connections to the protected service. Only after the user has passed authentication inside the SPA process does the application interact with the firewall's APIs to add an allow rule for new connections from the specified source IP to the service port for some predetermined time. There is no other form of feedback from the server to the client in terms of whether the authentication was successful or not. After this preallocated time has passed, the firewall rule will be removed by the process, and the state of the machine will return to passive mode. In case a session was established with the service in this window, the firewall should allow it to persist. A high-level overview of the described workflow is provided in Figure 2.

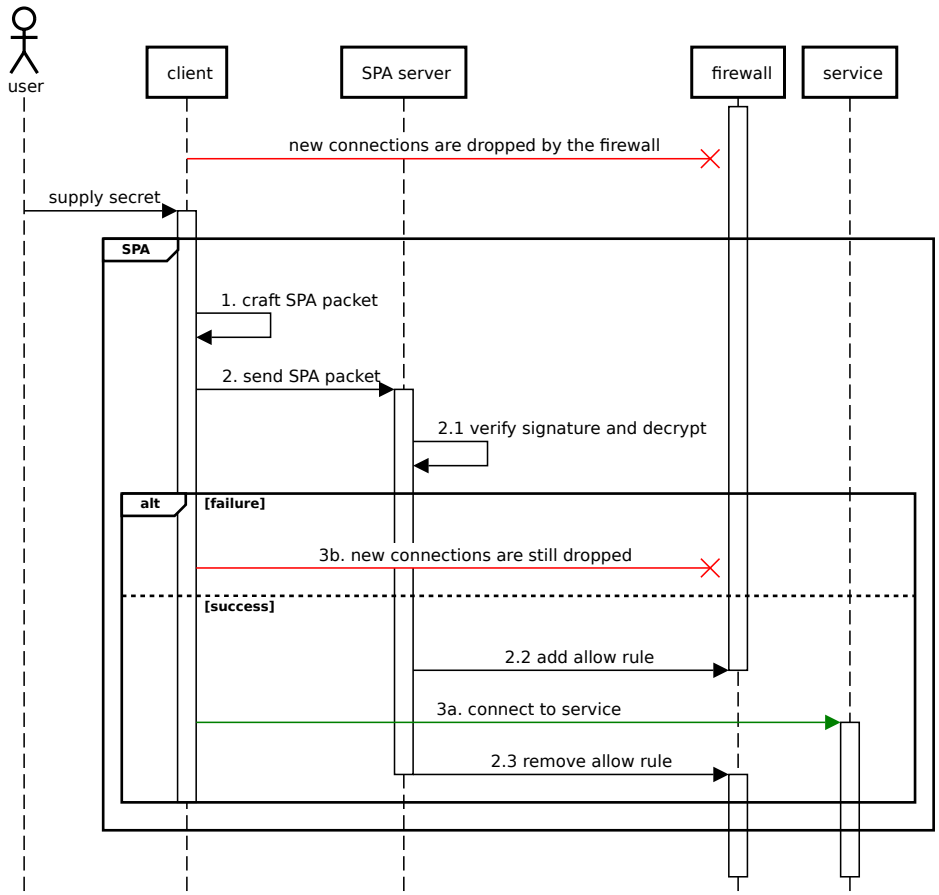

Figure 2. SPA workflow overview.

The term ”Single Packet Authorization” (SPA) was first coined by MadHat Unspecific and Simple Nomad at the BlackHat Briefings already back in July of 2005 [3]. However, there is still no uniform understanding or standardization that would define all the details of SPA; it is merely a concept of a protocol, which does not set many restrictions on the specific implementation. Due to the lack of central documentation, the SPA protocol has become a synthesis of multiple previous works, discussions, and concrete implementations. Nevertheless, the goal of this chapter is to summarize the idea of the protocol by combining information from all available sources.

2.1.1 Packet format overview

This subchapter explains the first step of the SPA workflow, namely how the packet is assembled and what it contains. MadHat and Nomad originally suggested [3] the authorization packet to consist of an identity token and session keys for authorization, a timestamp to counter replay attacks, and command data to allow access to specific services. After assembling the packet structure, it is then encrypted, and the ciphertext is signed with the server's GnuPG[6], or also known as GPG public key to provide both confidentiality and integrity for the data. The assembly process of an SPA packet is shown in Figure 3.

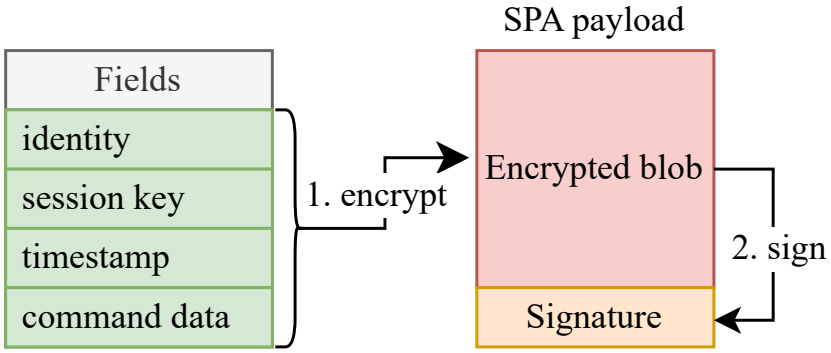

Figure 3. SPA packet format.

[6]https://gnupg.org/

This specific combination of encryption and signing follows the conventional encrypt-then-authenticate[7] paradigm that has also been used in the implementation of Transport Layer Security (TLS), for example. In order to check the provided signature and decrypt an incoming SPA packet, the server must be in possession of the corresponding GPG private key. An important remark that they make during the presentation is that the actual authentication happens when the signature is successfully verified and the decryption succeeds; there is no other authentication material within the decrypted packet. Following that is the authorization step, where the obtained data can be used for verifying that the requested access should be granted. This allows for fine-grained access control by defining that specific users can only access a subset of services available on the server.

2.1.2 Network transmission

After the packet is assembled by the client, it needs to be transported to the server in the second step. Even though the original SPA protocol description [3] defines TCP, UDP and/or ICMP for the transmission protocol, then the documentation for a specific implementation called *fwknop* [4], states that the packet can theoretically be delivered to the system using any network layer protocol of the OSI model[8]. In this case, the SPA message has to be encapsulated in some section of the protocol and defined on both the client and server applications. As an example, in case ICMP is used for transferring the data, the encrypted SPA payload can be stored in the data section of the packet as shown in Figure 4.

[7]https://www.rfc-editor.org/rfc/rfc7366#section-3
[8]https://learningnetwork.cisco.com/s/article/osi-model-reference-chart

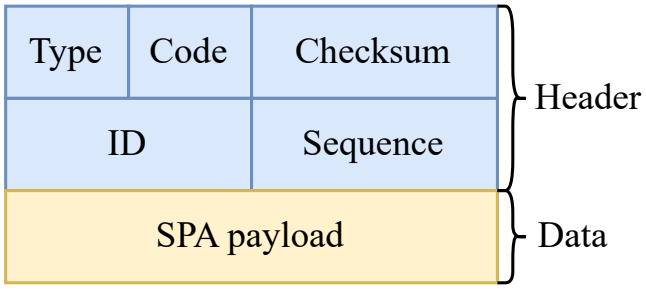

Figure 4. An SPA blob encoded into the data field of an ICMP packet.

For the server to receive the data, instead of opening a port, the process hooks into the operating system's low-level APIs to intercept incoming network traffic in kernel space as soon as it comes off the network card. The approximate packet movement starting from the transmission, extending to the SPA server and eventually reaching the service(s) can be seen in Figure 5.

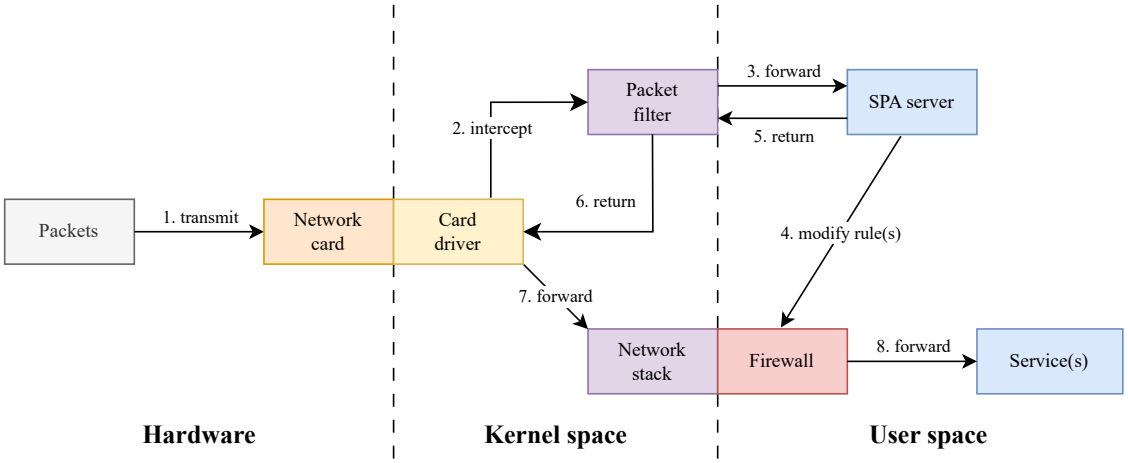

Figure 5. SPA operating layer. [5]

Doing so allows for processing the packets before they even get processed by the network stack itself, which makes further routing decisions and eventually passes the traffic on to the firewall. This is the core mechanism that allows for keeping the system firewall in a drop-all state, while still enabling SPA to function. Operating at such a low level has significant upsides in terms of possible use cases that are discussed in Section 2.1.4, but it also comes with some considerations and downsides, which are discussed next.

2.1.3 Limitations

The first thing to keep in mind when planning to use SPA is that, by definition, it requires privileged access to the server. Sniffing network traffic on a very low level, and interacting with the system firewall, is not possible otherwise. Running custom-crafted code with extensive permissions is not an issue, but allowing external applications full access to the underlying system requires extreme caution from the user. In addition, due to this interaction, the SPA server is entirely OS- and firewall-dependent, which makes a specific implementation rather immobile. Usage with exotic operating systems or firewalls most likely requires writing some additional integration code.

The main downside of lacking a standardized SPA specification means that many of the publicly available server and client applications may not be compatible with each other. This also gives way to basic implementations with inherent security concerns; one such case will be scrutinized in Section 3.1.

Using SPA in highly secured networks is generally possible due to it only using a single network packet, thus having a fairly low detection rate. Some network intrusion detection systems (IDS) may flag the packet, but without having access to the encryption key, the packet is just a blob of ciphertext that reveals nothing to the viewer. Network layer protocols of the OSI model like ICMP do not leverage ports for data transmission, making it harder for the IDS to specifically block SPA packets, unless all ICMP traffic is rejected. SPA also provides a possible solution for using it across networks with Network Address Translation (NAT) in place by supplying the desired source address inside of the packet data.

2.1.4 Use cases

This subchapter starts by exploring common use cases for SPA, followed by more sophisticated examples that test the limitations of such software. The current state of SPA is mostly meant for *ad hoc* use cases, without any concrete examples of widespread usage. Nevertheless, the fact that the SPA server intercepts packets within kernel space makes way for some creative applications that would not be possible otherwise.

Before diving into the possible use cases, a much-needed distinction which MadHat makes in his presentation [3] is that SPA is not a substitute for the service authentication by any means; it is an additional security layer that is independent of the underlying service.

In this sense, it can be compared to an additional authentication step that takes place before the firewall packet filtering.

In general, SPA is most useful in cases where access to the service is only needed occasionally, and security is a high concern. Therefore, the most obvious application of SPA is for device administration purposes in untrusted networks. Compared to other solutions that serve the same cause, SPA is much simpler to set up as it solely relies on a client and server application. There is no need to make any modifications to the surrounding infrastructure for SPA to function correctly. It also makes no distinction about the underlying service(s) that are being concealed, allowing for securing any type of service as desired.

The unique operating layer of SPA gives rise to an interesting use case with the concept of *ghost services* [6]. This concept makes use of NAT prerouting in order to redirect the arriving packets to a service that is concealed by an already exposed port. As an example, by default, the 192.168.0.1 machine exposes a web service Apache on port 80, but after a successful SPA authentication, all packets arriving to the machine from the SPA client's IP 172.16.0.1 to the destination port 80 will instead be redirected to 192.168.0.3 port 8080 where the admin panel exists. This leaves the original service still accessible to all other users while also concealing the usage of the admin panel for the authorized user.

As is the case with many security-related software and tools, they can be used for both defensive and offensive purposes. 0xLAITH explores in his paper [7] the possibility of using SPA for hiding backdoors while also providing a simple proof of concept. A backdoor is a malicious piece of software that is used by threat actors for maintaining access to a compromised host. The proof of concept script he created listens for an SPA packet and sends a reverse shell to a specified IP address after successful verification and decryption of the data. He concluded that the backdoor concept could be elaborated upon by attempting to hide the SPA process itself on the compromised machine.

3. Theory and Analysis

Therefore, using SPA allows relying on the system firewall for protecting the underlying services. This also means that the attack surface is now directed at SPA itself, making it crucial to examine that next.

The overall security of SPA is a complex subject involving many different aspects and related security assumptions. Possible attacks against it can be divided mainly into two distinct categories—network (passive and active) and cryptographic attacks. Some of the risks associated with either category are related to the underlying concepts of the protocol, while others are important only when dealing with a specific practical implementation. No matter the attack specifics, the end goal is to obtain the secret key material in order to further enumerate the hidden service(s).

Crafting a valid SPA packet requires the encryption and HMAC keys from the client, which are commonly stored in a single file on the same device. In the event of a system compromise, the adversary would obtain both of these secrets. Even if the secrets are not directly leaked, a network-level attacker can attempt to crack them offline from an eavesdropped SPA packet.

To remove the risk of a single point of failure, the Multi-Factor Authentication (MFA) schema can be used with a One-Time Password (OTP) system, which leverages numerical codes generated on a secondary device and verified on the server.

3.1 Related Work

Before moving on to the implementation details and choosing the most reasonable OTP system for usage with SPA, it is reasonable to study work that has already been done in this direction.

CSPAuthD[9] extends SPA to use Public-Key Infrastructure (PKI) for user authentication. While the usage of asymmetric cryptography is, in general, a great solution to the static secret problem and definitely helps against eavesdropping attacks, setting up a working PKI is a cumbersome process and not ideal for small-scale uses. On a smaller scale, GPG keys can still be used for the signature and encryption parts. At the end of the day, this

[9]https://xmit.xyz/software/cspauthd/

solution still leaves a single point of compromise, as the leakage of the private key allows for full access to the server.

Packet Verification System[10] is most closely related to the purpose of this thesis, because it also attempts to integrate a one-time password system with SPA. However, there are numerous security and ease-of-use concerns with the practical approach taken by the developer. Studying the source code reveals that instead of encrypting the SPA payload, it sends the raw SPA data through the socket[11] on the client side, and only uses a HMAC signature to verify the integrity of the received packet[12] on the server. The one-time code itself is appended to the payload in plaintext[13]. This leaves the implementation vulnerable to numerous different network-level attacks. For example, a Man-in-the-Middle (MitM) attack would be possible by dropping the sent packet and using the obtained numerical code to send a custom packet. A passive network attacker with sufficient time could also gather a significant amount of such codes and attempt a brute-force attack to obtain the initial secret like shown by unix-ninja[14]. Furthermore, since this implementation uses a completely different format for the SPA packet, it is not compatible with other server applications. As an additional pitfall, the repository itself only has a Python server, meaning that a full install of Python would be required on the IoT device to use this tool out of the box. Converting the server script to MicroPython[15] may be possible, but it may lack some of the used standard libraries.

3.2 One-Time Password (OTP)

Historically, pin code calculators have been used by banks and other facilities to implement a very similar MFA system with a single-use password. While mobile phones have replaced pin code calculators for the most part, modern-day algorithms still generate one-time

[10]https://github.com/l3enj/SPA/tree/main

[11]https://github.com/l3enj/SPA/blob/5b97d274838ed1bf6d126dd5228e63856e2498b3/client/client.py#L91C1-L94C45

[12]https://github.com/l3enj/SPA/blob/5b97d274838ed1bf6d126dd5228e63856e2498b3/server/server.py#L67C1-L73C21

[13]https://github.com/l3enj/SPA/blob/5b97d274838ed1bf6d126dd5228e63856e2498b3/client/client.py#L45C1-L53C6

[14]https://www.unix-ninja.com/p/attacking_google_authenticator

[15]https://micropython.org/

passwords on much of the same principles. One standardization for a one-time password system is given in RFC 2289 [8].

An OTP system comes with numerous advantages to static secrets. As described in the security considerations section of RFC 2289 [8], an OTP system is mostly meant to protect against passive network attacks; it does not provide any significant defense against an active attacker. Assuming a correct implementation, a single OTP is only usable within a short time-window, making the system resistant to replay attacks that could be carried out by an adversary eavesdropping on the traffic. Other than network attacks, more advanced OTP systems also provide protection against the infamous shoulder surfing technique. Finding a one-time password from a command line history file or gathering it from a keylogged machine may also turn out to be useless for an adversary if not leveraged quickly enough. Static passwords, on the other hand, may be left unchanged for years and could also provide access to multiple accounts or even different services due to password reuse.

While the security against network attacks is improved, most OTP systems require the server to store the initial secret in a recoverable form. This means that if OTP is the only authentication mechanism and an adversary gains access to the server's database, they would immediately be able to compromise all user accounts. Compared to passwords that are generally stored as a hash on the server in a database, to prevent attackers from gaining access to plaintext passwords in case the system was compromised. There is no need to store the plaintext password for validation; instead, the input is hashed and compared to the stored hash value. Luckily, the OTP system is most frequently used as an additional authentication factor, rarely ever as the only one.

3.2.1 HMAC-based One-Time Password (HOTP)

One of the simplest algorithms for a one-time password system is achieved in the implementation of HMAC-based One-Time Password (HOTP) that was first described by IETF RFC 4226 [9] approximately 20 years ago. The underlying key material consists of two parts - a shared secret K and an incremental counter C. The Hash-based Message Authentication Code (HMAC) function is used in conjunction with a hashing function, after which the output is truncated in order to obtain a fixed-length numerical code of 6-8 digits, which is the one-time password value:

$$\text{HOTP}(K, C) = \text{Truncate}(\text{HMAC-SHA-1}(K, C)),$$

Because the underlying secret stays the same, the client and server also need to keep track of the counter due to its purpose of rotating the secret after each use. Due to hashing being a non-correlatable function, altering the counter by a single value changes the obtained HOTP to an incomparable state. This means that a division in the counter values on either side renders the algorithm useless until the counters are resynchronized.

Using the HOTP system solves the single static secret issue, but as shown by O. Bölin in his Bachelor's thesis [10], the system itself is vulnerable to a brute-force attack. Using 1000 concurrently brute-forcing machines, he concluded that there is a 60% chance of guessing the HOTP for a fixed counter value in 10 minutes. This is possible because the rotation of the HOTP does not have any time factor involved; the counter is incremented only after a successful use.

3.2.2 Time-based One-Time Password (TOTP)

The so-called Time-based One-Time Password (TOTP) system described in RFC 6238 [11], is an extension to the classic HOTP system that aims to deal with the slow secret rotation issue. In essence, the algorithm still relies on the usage of HOTP with a shared static secret, but uses the current Unix time in place of the counter to rotate the obtained one-time password value in a predefined time-step window as follows:

$$\text{TOTP} = \text{HOTP}\left(K, \left\lfloor \frac{\text{current Unix time} - T_0}{X} \right\rfloor\right),$$

where X is the time step window, T_0 is the initial counter time (usually 0), and the current Unix time is widely known to be defined as the number of seconds since midnight UTC of January 1, 1970. Due to the floor division operation, the obtained TOTP remains constant in one time window, but is still rotated every X seconds even if the system is not being used at the time. In practice, the default value used for X is 30 seconds, which is well enough for allowing the user to read and write the TOTP, and for the authentication parties to communicate this value. However, it is possible for the client and server to choose any suitable value for the division as long as it is synchronized between the systems. A delay window can be used to account for cases where the client's timestamp during sending the packet does not fall into the same time-step window as the server's calculation.

As described in the security considerations section of the TOTP RFC 6238 [11], it is best to use a short time-step window and a maximum delay window of one time-step in order to reduce the possibility of an adversary brute-forcing a single TOTP code.

Validation of an incoming TOTP on the server is done by performing the same calculation to obtain the same code as shown in Figure 1.

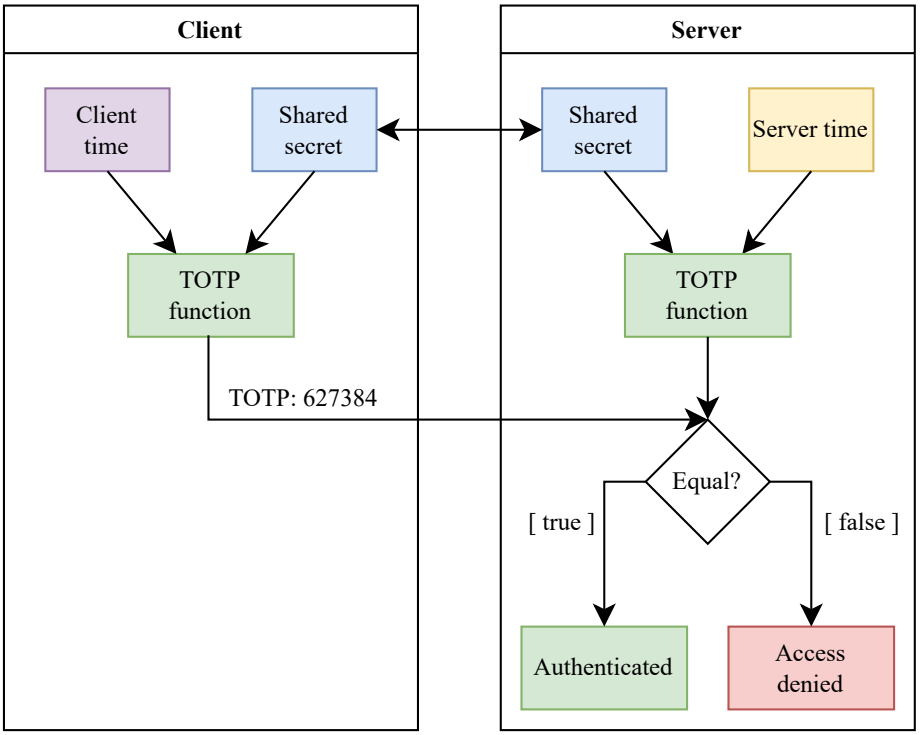

Figure 6. TOTP validation scheme. [12]

As such, using this system requires both sides to have access to the shared static secret and a loosely[16] accurate time source. Manipulation of the device's time source on either side leaves the system vulnerable to a forward-replay attack as shown by G. Bianchi and L. Valeriani [13]. This attack assumes full control of the time source and centers around the fairly straightforward concept of precalculating TOTPs for future timestamp values. After resetting the time source to the correct value, it is possible to use the stored TOTPs as the correct time instants arrive. As an example, the Google Authenticator[17] mobile

[16]Depending on the chosen X value.
[17]https://support.google.com/accounts/answer/1066447

app relies on the device's clock, not an external authoritative time source, allowing anyone to test this concept in a very simple fashion.

3.3 Proposed Solution

Both of the discussed algorithms provide a way to integrate the OTP system with already implemented software. In the case of SPA, the more reasonable choice is the TOTP system due to its time-based rotation system. In addition to that, over the last two decades MFA has become a prevalent security measure in online applications, which most commonly uses the described TOTP system [14], making it even more favorable over HOTP. Widespread usage of it has also led to a surge in authenticator applications such as Google Authenticator that can be installed on the end-users' mobile devices. These types of apps require the user to supply the initial secret and configuration parameters (such as the specific X), and can be used to generate TOTP codes. Storing the initial secret in the mobile application and using it for generating the codes instead of calculating them locally provides another authentication factor in the form of a second device.

4. Implementation

The practical part of this thesis involved integrating the Time-based One-Time Password (TOTP) system with the *fwknop* software. The primary objective of the integration was to provide a way to use TOTPs as an additional factor for user authentication. In terms of the architecture of the integration, the aims were to:

Goal 1. keep TOTP as an optional feature on the client;

Goal 2. make TOTP codes user-specific on the server;

Goal 3. ensure that any authenticator application can be leveraged for generating TOTP codes by the user;

The first goal guarantees that the server will remain compatible with other client applications that do not have TOTP implemented.

This chapter is divided into three main subchapters that can be shortly summarized as follows:

- Chapter 4.1: Overview of the *fwknop* implementation specifics;
- Chapter 4.2: Technical details about the development environment, added code and workflow modifications for the TOTP integration;
- Chapter 4.3: Results and validation for the work.

The lack of a concrete protocol definition makes it necessary to discuss the specific implementation of SPA in *fwknop* before proceeding to make modifications to the application source code.

4.1 Overview of fwknop

FireWall KNock OPerator (*fwknop*) is an open-source implementation of SPA written entirely in the low-level programming language C. It has minimal library dependencies, which is achieved by having a custom-made library *libfko* for the required cryptographic operations. The tool currently has built-in support for *iptables*, *firewalld*, PF, and *ipfw* firewalls across the Linux, OpenBSD, FreeBSD, and Mac OS X operating systems, as mentioned in the GitHub source code repository. [15]

There are multiple reasons for choosing *fwknop* over other available tools:

1. implementation in a low-level language makes it more difficult to modify, but the tool already comes with a significant amount of features;
2. the C programming language makes it easy to use on microcontrollers and other IoT devices;
3. the tool has already been reviewed in previous research [2].

4.1.1 Implementation specifics

An SPA packet for *fwknop* mostly follows the format that was described in Section 2.1.1, together with the added fields of random data bytes (protection against replay attacks), the *fwknop* version (used for compatibility reasons), and the message type as stated by the *fwknop* documentation [4]. Identity was renamed to username, and the session key field was removed altogether in the implementation. The message type field allows the client to specify if they aim to just add a basic rule to the firewall, or gain NAT access to an internal service as was demonstrated with the *ghost services* concept. In addition to the mandatory packet fields, *fwknop* also comes with some optional ones - NAT access request, third-party authentication information, and firewall rule timeout. Another added feature is the inclusion of the data digest inside of the packet. After assembling the packet, it is encrypted for confidentiality and the corresponding ciphertext is signed with HMAC to provide integrity. The approximate packet structure can be seen in Figure 7.

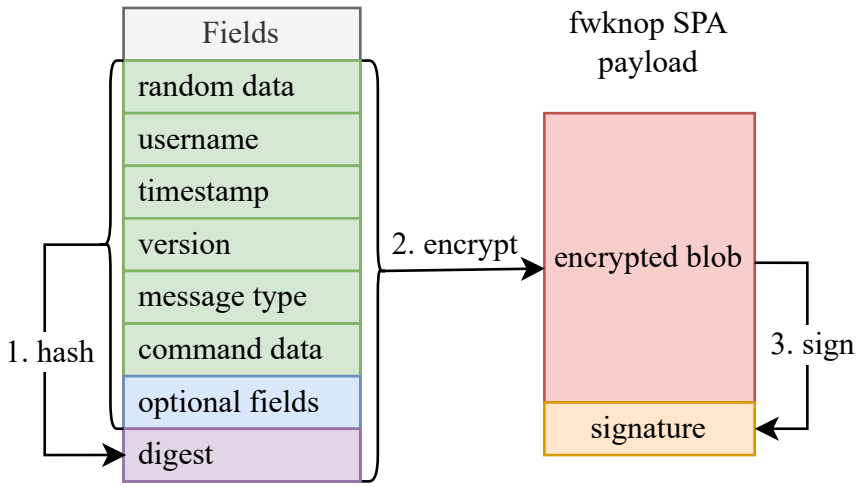

Figure 7. SPA packet format in fwknop.

Instead of relying on fixed lengths for fields, the data is assembled in a certain order by separating each entry with a colon (:) character. The optional fields are added last, with the server checking the exact format to determine how many of these fields are present and which ones exactly. Some of the fields are base64 encoded before the assembly in order to deal with non-printable characters.

At the time of this writing, *fwknop* allows for two primary methods of encryption—symmetric or asymmetric. The former leverages the Advanced Encryption Standard (AES) block cipher in CBC mode with a static secret, while the latter relies on the encryption keys provided by GPG. The required keys and other parameters can be supplied to the client application via command-line arguments or in the *.fwknoprc* configuration file. One possible example of this file is given in Listing 1, while explanations for all available fields can be found from the *fwknop* Linux manual page[18].

```
1 [spa-server]
2 ACCESS                    tcp/80
3 SPA_SERVER                172.18.0.2
4 ALLOW_IP                  172.18.0.3
5 KEY_BASE64                1OHsGO0jB+eRivGoNzec8g==
6 HMAC_KEY_BASE64           XMRP+hsgkx0wThsZ4gU1MGTTg9yd3USdNzPIFopdOyd1cS1m6Rx
7 USE_HMAC                  Y
8 VERBOSE                   Y
```

Listing 1. Example .fwknoprc file on the client.

An *fwknop* server is actually called *fwknopd*, where the added letter stands for the Linux concept of a *daemon*[19], meaning a background process. The *fwknopd* server relies on two different types of configuration files—the *fwknopd.conf* file deals with operational parameters for the server itself, while the *access.conf* file(s) are used for access control directives for each user. Depending on which encryption method is used, the configuration file should contain the required keys. An example of the *access.conf* file is given in Listing 2, which includes the encoded encryption key on line 2 and the encoded HMAC key on line 3.

[18]https://www.cipherdyne.org/fwknop/docs/manpages/fwknop.html
[19]https://www.man7.org/linux/man-pages/man7/daemon.7.html

```
1 SOURCE              ANY
2 KEY_BASE64          1OHsGO0jB+eRivGoNzec8g==
3 HMAC_KEY_BASE64     XMRP+hsgkx0wThsZ4gU1MGTTg9yd3USdNzPIFopdOyd1cS1m6Rx
```

Listing 2. Example access.conf file on the server.

4.1.2 Packet decryption process

Studying the source code reveals that depending on the underlying operating system and startup parameters, *fwknopd* gains access to network traffic before the firewall by using the APIs exposed by either *netfilter* or *libpcap*. Essentially, *libpcap* is a library that allows obtaining the raw packet bytes from the Linux kernel as soon as they come off of the network card. A network filter can be used in order to limit the amount of received packets. In the case of *fwknop*, the low-level C programming language is used to interact with the API, for which the definitions can be found in the *libpcap/pcap/pcap.h*[20] header file available in the GitHub open-source repository.

Thoroughly processing every single packet that the system receives could cause performance issues and leave the system vulnerable to a trivial Denial of Service (DoS) attack. In order to avoid this, specific network traffic filters can be used and prechecks can be done on the received packets, leaving any irrelevant packets untouched. Upon receiving a matching packet *fwknopd* first does some prechecks on the data format, in order to discover whether it is an SPA packet at all. After which, it loops through all the available access directive contexts and tries to decrypt with either Rijndael or GPG depending on the keys that the file specifies. This follows the same concept that was originally meant for SPA, where a successful decryption of the received blob is the authentication itself. No other authentication material is transmitted with the packet. After the verification, decryption, and parsing, the obtained variables can then be used for making authorization decisions. A high-level overview for validating an SPA packet on the *fwknopd* server is presented as Algorithm 1.

[20]https://github.com/the-tcpdump-group/libpcap/blob/master/pcap/pcap.h

Algorithm 1 fwknop SPA packet validation pseudocode

```
1: Read incoming packet into spa_pkt.
2: Read each access control file into access_stanza list.
3: function INCOMING_SPA(spa_pkt, access_stanza[ ])
4:     if precheck_spa_packet(spa_pkt) is not successful then
5:         return
6:     end if
7:     for acc in access_stanza[ ] do
8:         if verify_HMAC(spa_pkt, acc.hmac_key) is not successful then
9:             continue
10:        end if
11:        if rijndael_decrypt(spa_pkt, acc.key) is successful then
12:            spa_data ← parse(spa_pkt)
13:        end if
14:        if gpg_decrypt(spa_pkt, acc.key) is successful then
15:            spa_data ← parse(spa_pkt)
16:        end if
17:        if authorization_checks(spa_data, acc) is not successful then
18:            continue        ▷ Checks for example, supplied username, and source IP
19:        end if
20:        return spa_data
21:    end for
22:    return
23: end function
```

4.2 TOTP integration

A Docker compose environment with separate containers for the *fwknop* server and client applications was developed to allow for easy deployment and testing of the solution in a constrained environment.

The development environment is a fork of a publicly available example environment[21], which was extended for the specific needs of this thesis and is available for review in the Appendix section. Making changes in the local fork of the application source code was synced into the containers with the usage of a Docker volume[22]. Instead of rebuilding both of the containers after each modification in the code, a Bash script was used inside the containers for recompilation and subsequent configuration changes.

In order to remain compatible with available authenticator applications (e.g. Microsoft Authenticator) the RFC 6238[23] was followed throughout the whole implementation process. This allows the user to leverage any authenticator application that they find suitable for generating the TOTP codes.

The required code for the practical part was written by following the GNU Coding Standards[24] in order to keep the code clean and remain uniform with the existing *fwknop* source code. All added files were supplied with a brief Doxygen[25] description; inline comments were added where necessary. Debugging of the code was done with The GNU Project Debugger (GDB)[26], and its GEF[27] extension meant for advanced usage.

Due to the lack of existing Windows-based server applications, the final product was developed only for Unix-like systems. The *iptables*[28] firewall was used, as it is the default selection for *fwknop* due to its simplicity and widespread usage in servers, though the created TOTP integration is independent of the firewall.

4.2.1 Additional access control directives

An optional command-line flag *–totp* was added to the client binary, which opens a non-graphical prompt to obtain the TOTP code from the user.

Since the initial secret needs to be stored on both the client and server applications, then it has to be generated on one side and transferred over. Base32 uses a character set of

[21]https://github.com/antoniopaya22/SPA-Example
[22]https://docs.docker.com/engine/storage/volumes/
[23]https://www.rfc-editor.org/rfc/rfc6238
[24]https://www.gnu.org/prep/standards/standards.html
[25]https://www.doxygen.nl/index.html
[26]https://sourceware.org/gdb/
[27]https://github.com/hugsy/gef
[28]https://linux.die.net/man/8/iptables

uppercase ASCII letters (A-Z) and a subset of numbers (2-7), which has made it the standard for TOTP initial secret encoding due to it being easy for humans to read, and write to an authenticator application.

In order to provide the required initial secret for the server, the *TOTP_KEY* and *TOTP_KEY_BASE32* variables were added to the *access.conf* file specification. These parameters follow the same naming scheme as for Rijndael, where only one of them is required, but the encoded version allows for keeping the initial keyspace generated by */dev/urandom*, without filtering out non-printable bytes.

4.2.2 Extending the libfko library

Instead of relying on an external cryptographic dependency like OpenSSL, the *fwknop* tool has its own custom-made *libfko* library. It is quite extensive in its functionality and offers methods for multiple hashing algorithms as well as HMAC functions which are sufficient for the purpose of this thesis. To avoid duplicate code and other dependencies, it is possible to leverage the already defined functions for the purpose of TOTP generation on both the client and server applications.

Initial secret generation for the TOTP algorithm was done using the existing *get_random_data* function, which leverages the */dev/urandom* Linux special file for obtaining cryptographically secure pseudorandom data. Due to the usage of HMAC-SHA1, the length for the initial secret was chosen to be 20 bytes (i.e. 32 base32 characters), which is in accordance with the security considerations explained in Section 5 of RFC 6238[29].

The *libfko* library was extended with files containing the definitions for the *fko_totp_from_secret*, *fko_base32_encode* and *fko_base32_decode* functions. The former allows for generating a TOTP code with a provided initial secret and timestamp. Simplified pseudocode for the function is given in Algorithm 2.

[29]https://www.rfc-editor.org/rfc/rfc6238#section-5

Algorithm 2 Generating the current TOTP from an input secret

1: Read *totp_secret* from *access.conf* file.
2: Store the current Unix time in *timestamp*.
3: **function** FKO_TOTP_FROM_SECRET(totp_secret)
4: digits $\leftarrow 6$ ▷ desired length of the obtained code in range 6 to 8
5: $X \leftarrow 30$ ▷ default time-step size
6: $T \leftarrow \lfloor \frac{timestamp}{X} \rfloor$
7: $hmac\_result \leftarrow$ HMAC_SHA_1($totp\_secret, T$)
8: $totp\_code \leftarrow$ TRUNCATE($hmac\_result$) mod 10^{digits}
9: **return** *totp_code*
10: **end function**

The latter two are utility functions implemented according to RFC 4648[30], that are used in the key generation and storage process. Numerous other utility functions were created in order to provide the SPA process access to the TOTP inside the packet, and populate the obtained authorization data after the decryption succeeds.

4.2.3 Transmitting and verifying the TOTP

For the authentication to succeed, the client needs to first calculate the TOTP with the newly defined function, followed by transferring it to the server along with the rest of the SPA data. One way for achieving this is by storing the TOTP inside an optional field in the SPA packet.

As mentioned previously, an SPA packet in *fwknop* has an optional field called third-party authentication. Initially, this sounds like a fairly good candidate for storing the TOTP, but the existing documentation lacks any mention of this field. Looking through the source code reveals that this field is actually considered legacy and was used for server authentication. Currently, the client application does not even have an option for using it. Any other fields are not viable for the TOTP storage, leading to the creation of a completely new optional field.

[30]https://www.rfc-editor.org/rfc/rfc4648

If the verification and decryption of the SPA packet succeed on the server, and the access control file contained either of the newly added control directives, an additional check is done on the contained TOTP code before granting the user any further access. The TOTP check could be improved upon by implementing a throttling parameter as explained in RFC 6238 [11], which would limit the amount of times a single TOTP code can be used in a specific time-step window.

4.3 Results

To summarize, as part of writing the TOTP integration, a Docker environment was built consisting of two separate containers to allow for testing the solution. Modifications made to the *fwknop* source code included:

- creation of library functions for TOTP calculation, initial secret generation, and base32 encoding/decoding utilities;
- the addition of an optional TOTP field inside the SPA packet;
- the inclusion of an optional command-line flag for the client to use this field;
- changes to the server's workflow to verify the acquired TOTP.

This subchapter provides an example walkthrough for using the added functionality and gives an overview of how the written code was validated.

4.3.1 Example usage

To leverage the added TOTP functionality for additional authentication, the first step is to generate a new pair of keys on the server with the *–key-gen* flag as seen in Figure 8.

```
root@f1786f1f97f1:/gateway# fwknopd --key-gen
KEY_BASE64: uZQQSOBvlhH2KArcmOfmeQ8CjDXkgdP4uHppc4uPTjY=
HMAC_KEY_BASE64: El6Tt6IJ7nklSebJDyDKnfdZGEx80yXXDmqopamlQxwCsSNwqoaP4Mn5omU57qmGArTF0z8RfRn4GvoyxrdhnA==
TOTP_KEY_BASE32: GNE3FOGV3YUC2FFJHWM4UENZKIT44PVW
root@f1786f1f97f1:/gateway#
```

Figure 8. Generating the required keys on the server.

This results in the *access.conf* file being the following:

```
SOURCE                          ANY
KEY_BASE64                      uZQQSOBvlhH2KArcmOfmeQ8CjDXkgdP4uHppc4uPTjY=
HMAC_KEY_BASE64
↪ El6Tt6IJ7nklSebJDyDKnfdZGEx80yXXDmqopamlQxwCsSNwqoaP4Mn5omU57qmGArTF0z8RfRn4GvoyxrdhnA==
TOTP_KEY_BASE32      GNE3FOGV3YUC2FFJHWM4UENZKIT44PVW
```

Listing 3. The access.conf file on the server.

The TOTP key can comfortably be transported to an authenticator application by using some ASCII to QR code encoder, like the Python3 qrcode[31] package as seen in Figure 9, and scanning it from the mobile application.

Figure 9. Encoding the TOTP secret in a QR code for transportation.

After transferring the rest of the keys to the SPA client, and formatting them accordingly along with the source and SPA server IPs, the *.fwknoprc* file may look as shown in Listing 4.

[31]https://pypi.org/project/qrcode/

```
[spa-server]
ACCESS                      tcp/80
SPA_SERVER                  172.18.0.2
ALLOW_IP                    172.18.0.3
KEY_BASE64                  uZQQSOBvlhH2KArcmOfmeQ8CjDXkgdP4uHppc4uPTjY=
HMAC_KEY_BASE64
    ↪ El6Tt6IJ7nklSebJDyDKnfdZGEx80yXXDmqopamlQxwCsSNwqoaP4Mn5omU57qmGArTFOz8RfRn4GvoyxrdhnA==
USE_HMAC                    Y
```

Listing 4. Generated .fwknoprc file on the client.

Using *nmap* from the client to scan the server's port 80 shows it as filtered by the firewall as seen in Figure 10.

```
root@b03b15734eec:/client# nmap -p80 spa-server
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-05-14 13:44 UTC
Nmap scan report for spa-server (172.18.0.2)
Host is up (0.000056s latency).
rDNS record for 172.18.0.2: spa-server.spa-fwknop-testing_spa-net

PORT   STATE    SERVICE
80/tcp filtered http
MAC Address: 02:42:AC:12:00:02 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.43 seconds
root@b03b15734eec:/client#
```

Figure 10. Port 80 is filtered on the server.

Sending the SPA packet can now be done by issuing the command *fwknop -n spa-server –totp* and supplying the TOTP from the authenticator application into the prompt. The double verbose flags can be used to display the TOTP and encoded data as seen in Figure 11.

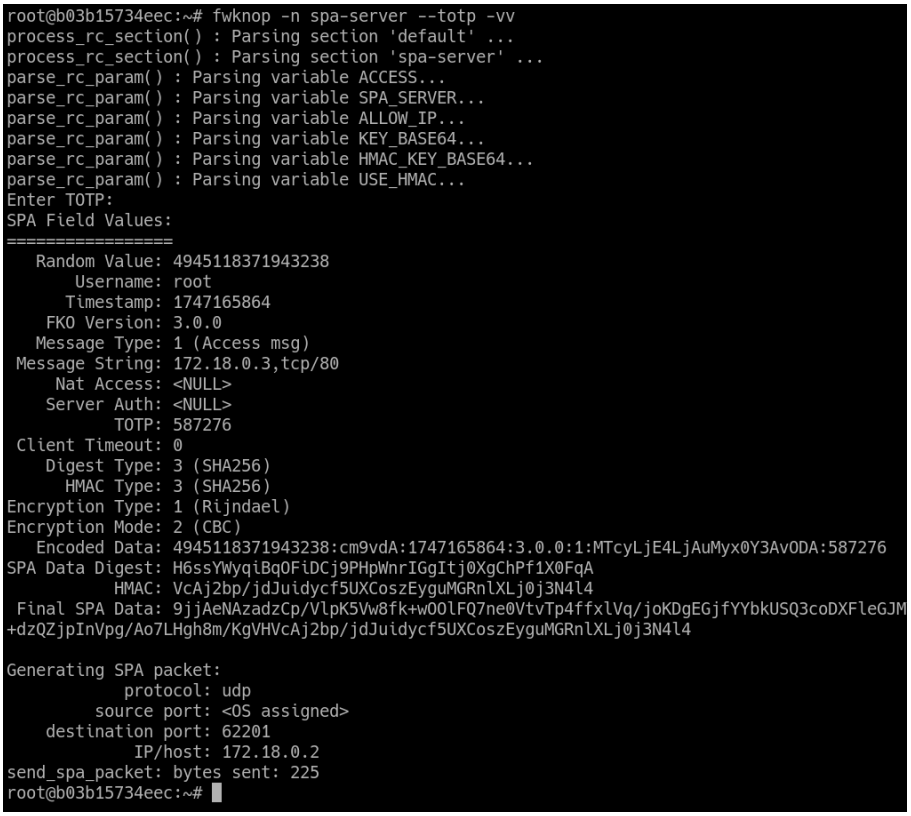

Figure 11. Client debug logs when sending TOTP.

Running the server in debug mode shows the received SPA packet, along with the decoded data, which includes the TOTP as shown in Figure 12. Since the verification, decryption, decoding and subsequent TOTP check were successful, the process will continue by adding a firewall allow rule to the requested service behind port 80.

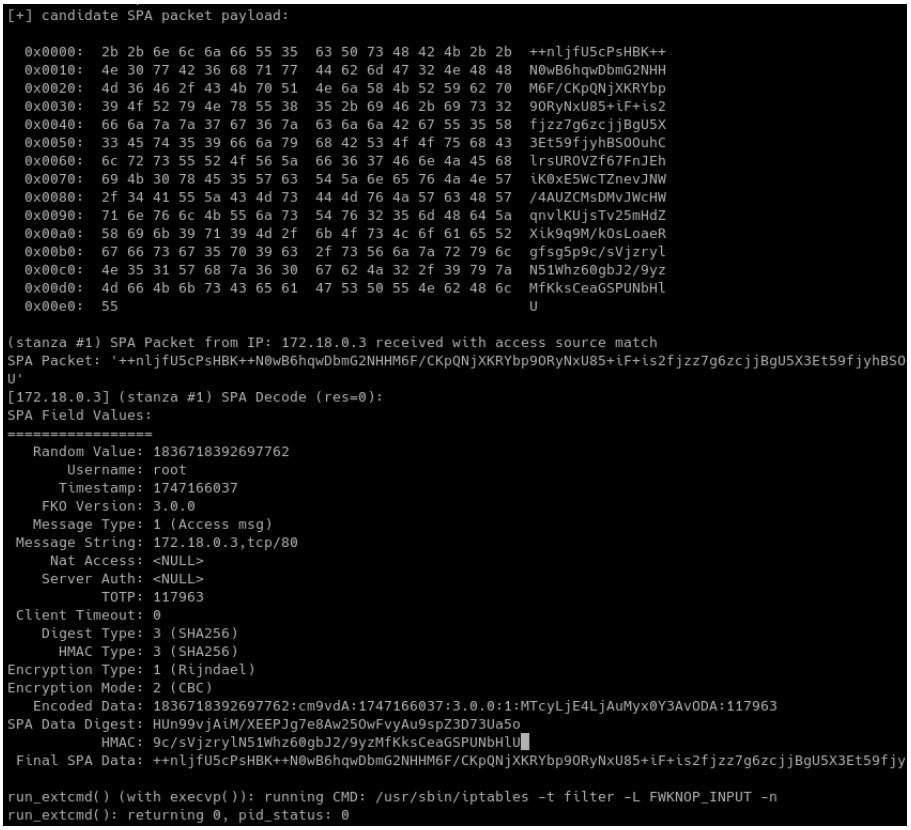

Figure 12. Server logs showing the received TOTP.

The client can now verify that the port is open by conducting another *nmap* scan against the server as seen in Figure 13.

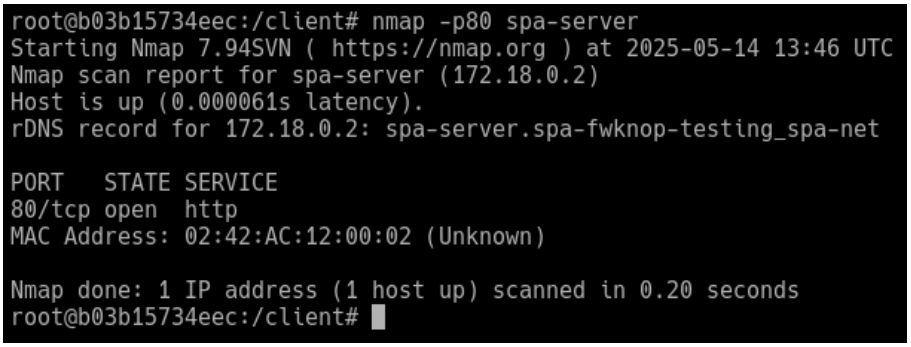

Figure 13. Port 80 is open on the server.

4.3.2 Validation

Functional validation was carried out on all the written code.

Unit tests were written for the base32 encoding and decoding functions using the CUnit framework[32], ensuring that the server and client obtain the exact same initial TOTP secret.

Validating the TOTP generation was done using common authenticator applications such as Google Authenticator[33] and Authy[34] for the code generation and comparing them to the server's calculations.

Another important part of functionality verification consisted of ensuring that the server can handle both SPA packets with and without the TOTP field. This also confirmed that a client without the TOTP functions implemented can still use the server in the exact same way as it was able to before.

[32]https://cunit.sourceforge.net/

[33]https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2

[34]https://www.authy.com/

5. Conclusion

This paper explored the necessity of Single Packet Authorization (SPA) as a simpler alternative to other measures for providing network-level protection for services. Compared to similar solutions, SPA is relatively easy to set up, and does not require modifications to the surrounding infrastructure.

The practical part of the thesis achieved Multi-Factor Authentication (MFA) with the Time-based One-Time Password (TOTP) system using the *fwknop* software. This required adding new access control directives for secret key storage, storing the TOTP in an optional field inside the SPA packet, and integrating the TOTP generation and subsequent checks into the client and server applications. All the goals that were set in the practical part were achieved, making the written solution an optional user-specific feature, and remaining compatible with authenticator mobile applications such as Google Authenticator.

Using the *fwknop* application with the crafted TOTP system alleviates the concern of a single point of failure that is related to shared static secrets. Even if a network-level attacker manages to crack the encryption and HMAC keys from an eavesdropped packet, they would not be able to craft a valid SPA packet, unless they are also in possession of the device with the TOTP secret.

Further development of the created integration could consist of implementing a throttling parameter on the server to limit the amount of times a TOTP can be used in a given time-step window.

References

[1] Mukherjee A. Network Security Strategies: Protect your network and enterprise against advanced cybersecurity attacks and threats. Packt Publishing Ltd, 2020.

[2] Jeanquier S. An Analysis of Port Knocking and Single Packet Authorization. Master's thesis. University of London, 2006.

[3] MadHat Unspecific S. N. SPA: Single Packet Authorization. 2005. https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-madhat.pdf (05/15/2025).

[4] Rash M. Single Packet Authorization with Fwknop. https://www.cipherdyne.org/fwknop/docs/fwknop-tutorial.html (05/15/2025).

[5] Garcia L. M. Programming with libpcap-sniffing the network from our own application. *Hakin9-Computer Security Magazine* 2 (2008), p. 2008.

[6] Rash M. Creating Ghost Services with Single Packet Authorization. 2009. https://cipherdyne.org/blog/2009/11/creating-ghost-services-with-single-packet-authorization.html (05/15/2025).

[7] 0xLAITH. Hiding a Backdoor with Single Packet Authorization. 2020. https://github.com/0xLAITH/SPA_Backdoor/blob/master/Hiding%20a%20Backdoor%20with%20Single%20Packet%20Authorization%20by%200xLAITH.pdf (05/15/2025).

[8] Haller N., Metz C., Nesser P., and Straw M. RFC 2289: A One-Time Password System. Tech. rep. Internet Engineering Task Force (IETF), 1998. https://www.rfc-editor.org/rfc/rfc2289 (05/15/2025).

[9] M'Raihi D., Bellare M., Hoornaert F., Naccache D., and Ranen O. RFC 4226: HOTP: An HMAC-Based One-Time Password Algorithm. Tech. rep. Internet Engineering Task Force (IETF), 2005. https://www.rfc-editor.org/rfc/rfc4226 (05/15/2025).

[10] Bölin O. and Van Daele P. Penetration Testing of One-Time Password Authentication. 2024.

[11] M'Raihi D., Machani S., Pei M., and Rydell J. RFC 6238: TOTP: Time-based One-Time Password Algorithm. Tech. rep. Internet Engineering Task Force (IETF), 2011. https://www.rfc-editor.org/rfc/rfc6238 (05/15/2025).

[12] Oliynyk M. TOTP Algorithm Explained. https://www.protectimus.com/blog/totp-algorithm-explained/ (05/15/2025).

[13] Bianchi G. and Valeriani L. Time is on my side: Forward-replay attacks to totp authentication. *International Symposium on Security and Privacy in Social Networks and Big Data*. Springer. 2023, pp. 109–126.

[14] Al-Sahli R. A., Al-Mutairi A. A., and Nasr K. Secure authentication system based on multi-factor authentication. *Taibah University* (2024). DOI: 10.13140/RG.2.2.24880.74247.

[15] Rash M. fwknop - Single Packet Authorization. https://github.com/mrash/fwknop (05/15/2025).

Appendices

I. Source code repositories

The source code for the Time-based One-Time Password (TOTP) integration can be found at: https://github.com/ricoandreaslepp/fwknop-totp.

The development environment code is stored in: https://github.com/ricoandreaslepp/SPA-fwknop-testing.

II. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Rico-Andreas Lepp,

1. grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis Enhancing Single Packet Authorization with Multi-Factor Authentication, supervised by Tarmo Oja.

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Rico-Andreas Lepp

15.05.2025