WHITEPAPER

Zerologon:

Unauthenticated domain controller compromise by subverting Netlogon cryptography (CVE-2020-1472)

by Tom Tervoort, September 2020

Secura

Zerologon:

Unauthenticated domain controller compromise by subverting Netlogon cryptography (CVE-2020-1472)

Summary

This whitepaper describes some of the technical details of CVE-2020-1472 (which we have dubbed "Zerologon"), a critical vulnerability in Windows Server that has received a CVSS score of 10.0 from Microsoft. In order to mitigate this issue, it is highly recommended to install Microsoft's August 2020 security patches on all Active Directory domain controllers. Leaving a DC unpatched will allow attackers to compromise it and give themselves domain admin privileges. The only thing an attacker needs for that is the ability to set up TCP connections with a vulnerable DC; i.e. they need to have a foothold on the network, but don't require any domain credentials.

The patch that addresses Zerologon also implements some additional defense-in-depth measures that forces domain-joined machines to use previously optional security features of the Netlogon protocol. An update in February 2021 will further tighten these restrictions, which may break some third-party devices or software. Please note that installing the August 2020 patch on all domain controllers (also back-up and read-only ones) is sufficient to block the high-impact exploit described here. Refer to Microsoft's guide on these changes for more information.

If you want to make sure you are not vulnerable, you can make use of the test-tool that we published and can be downloaded from our Github repo at **https://github.com/SecuraBV/CVE-2020-1472**. We will not release a complete working Proof-of-Concept exploit, but it is our assessment that such an exploit could be constructed by malicious actors with some effort, based on the patch for the CVE alone.

The attack described here takes advantage of flaws in a cryptographic authentication protocol that proves the authenticity and identity of a domain-joined computer to the DC. Due to incorrect use of an AES mode of operation it is possible to spoof the identity of any computer account (including that of the DC itself) and set an empty password for that account in the domain.

Vulnerability Details

The Netlogon protocol

The Netlogon Remote Protocol is an RPC interface available on Windows domain controllers. It is used for various task related to user and machine authentication, most commonly to facilitate users logging in to servers using the NTLM protocol. Other features include the authentication of NTP responses, and notably: letting a computer *update its password* within the domain. The RPC interface is available over TCP through a dynamic port allocated buy the domain controller's '*portmapper*' service, or through an SMB pipe on port 445.

What's interesting about this protocol is that it does not use the same authentication scheme as other RPC services. Instead it uses a customized cryptographic protocol to let a client (a domain-joined computer) and server (the domain controller) prove to each other that they both know a shared secret. This shared secret is a hash of the client's computer account password. The reason for this is that computer accounts did not use to be first-class principles in the Windows NT days, so they could not make use of standard user authentication schemes like NTLM or Kerberos.

A Netlogon session is initiated by the client, whereby client and server exchange random 8-byte nonces (called client and server *challenges*) with each other. They both compute a *session key* by mixing both challenges with the shared secret using a key derivation function. Then the client uses this session key to compute a *client credential*. The server recomputes this same credential value and if it matches it is concluded that the client must know the session key, and therefore the client must also know the computer password.

During the authentication handshake both parties can negotiate whether they want to seal and sign (encrypt and cryptographically authenticate) subsequent messages, which is essential to protect against network-level attackers. When encryption is disabled, all Netlogon calls that perform an important action must still contain an *authenticator* value that is also computed using the session key.

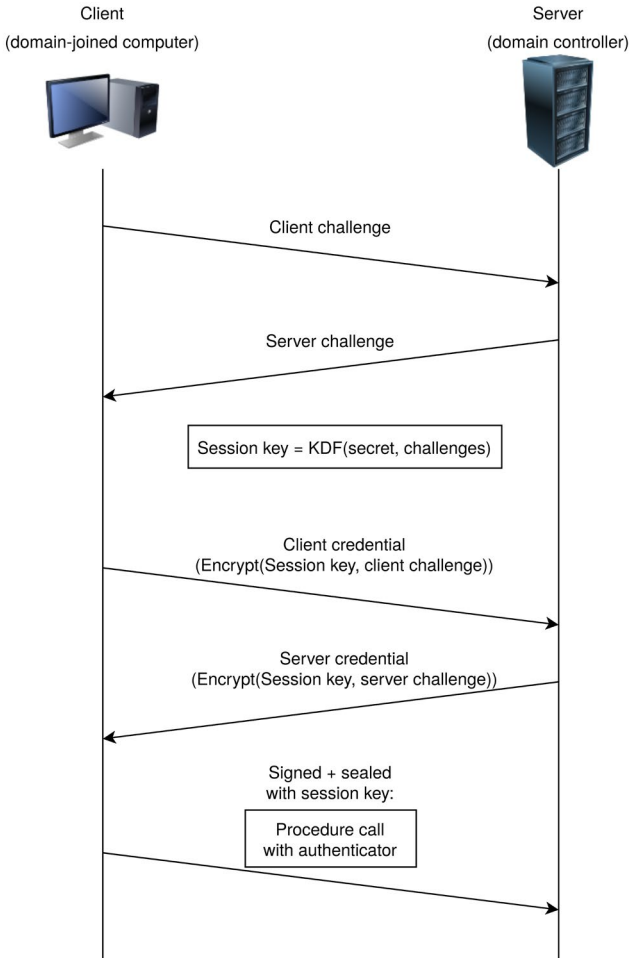

Figure 1: Simplified Netlogon authentication handshake

Implementing cryptographic protocols is tricky: one small oversight can lead to all kinds of methods to bypass the intended function of the scheme (in this case: computer authentication and transport security). Since I was not aware of any published security audits of this protocol, I decided to take a deeper look at it myself. Initially, I was mostly looking for person-in-the-middle attacks that assume an attacker that can see and modify traffic between a legitimate client and server. This yielded **CVE-2019-1424**, which could be used to gain local admin access to client systems of which the attacker could see and modify traffic. However, afterwards, when more closely examining the cryptography used for the initial authentication handshake, I discovered a much more severe general authentication bypass, which can be carried out by any attacker who is able to set up a TCP connection with the domain controller.

Core vulnerability: insecure use of AES-CFB8

The cryptographic primitive both the client and server use to generate credential values is implemented in a function called *ComputeNetlogonCredential*, as defined in the **protocol specification**. This function takes an 8-byte input and performs a transformation on it with the secret session key that produces an output of equal length. The underlying assumption behind it is that an attacker who does not know the session key will not be able to calculate or guess the correct output matching a certain input, allowing it to be used to prove knowledge of the session key.

There are two versions of this function: one based on 2DES and a newer version based on AES. Which one is used depends on flags set by the client during authentication. However, the default configuration of a modern version of Windows Server will reject any attempt to authenticate using the 2DES scheme. Therefore, in most domains only the AES scheme can be used. Interestingly, it is precisely this newer scheme in which I found the vulnerability. The older version is not affected by this specific attack (although 2DES is still considered insecure for other reasons).

The basic AES block cipher operation takes an input of 16 bytes and permutes it to an equally-sized output. In order to encrypt larger or smaller inputs a *mode of operation* has to be chosen. The *ComputeNetlogonCredential* function, which needs to transform only 8 bytes, makes use of the rather obscure **CFB8 (8-bit cipher feedback) mode**. This mode is about 16 times slower than any of the more common modes of operation used with AES, which probably explains why it is not widely used.

AES-CFB8 encrypts each byte of the plaintext by prepending a 16-byte 'Initialisation Vector' to the plaintext, then applying AES to the first 16 bytes of the IV+plaintext, taking the first byte of the AES output, and XOR'ing it with the next plaintext byte. This is illustrated in Figure 2.

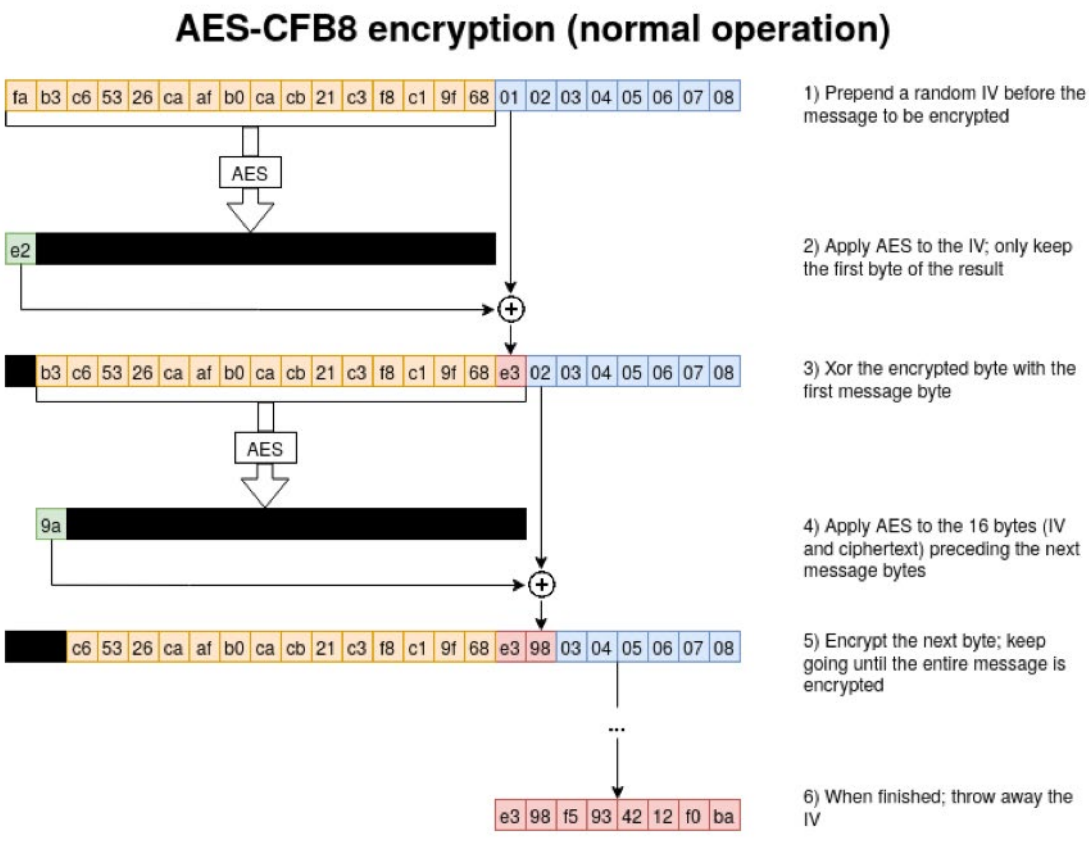

Figure 2: An illustration of encryption with the AES-CFB8 mode of operation.

In order to be able to encrypt the initial bytes of a message, an Initialisation Vector (IV) must be specified to bootstrap the encryption process. This IV value must be unique and randomly generated for each separate plaintext that is encrypted with the same key. The *ComputeNetlogonCredential* function, however, defines that this **IV is fixed and should always consist of 16 zero bytes**. This violates the requirements for using AES-CFB8 securely: its security properties only hold when IVs are random.

So, is this actually a problem here? What can go wrong with an all-zero IV? Because of the obscurity of CFB8, I could not find any literature on this subject. So I tried to come up with some *chosen-plaintext attacks* myself and figured out something interesting: **for 1 in 256 keys, applying AES-CFB8 encryption to an all-zero plaintext will result in all-zero ciphertext**. Figure 3 shows why this is the case.

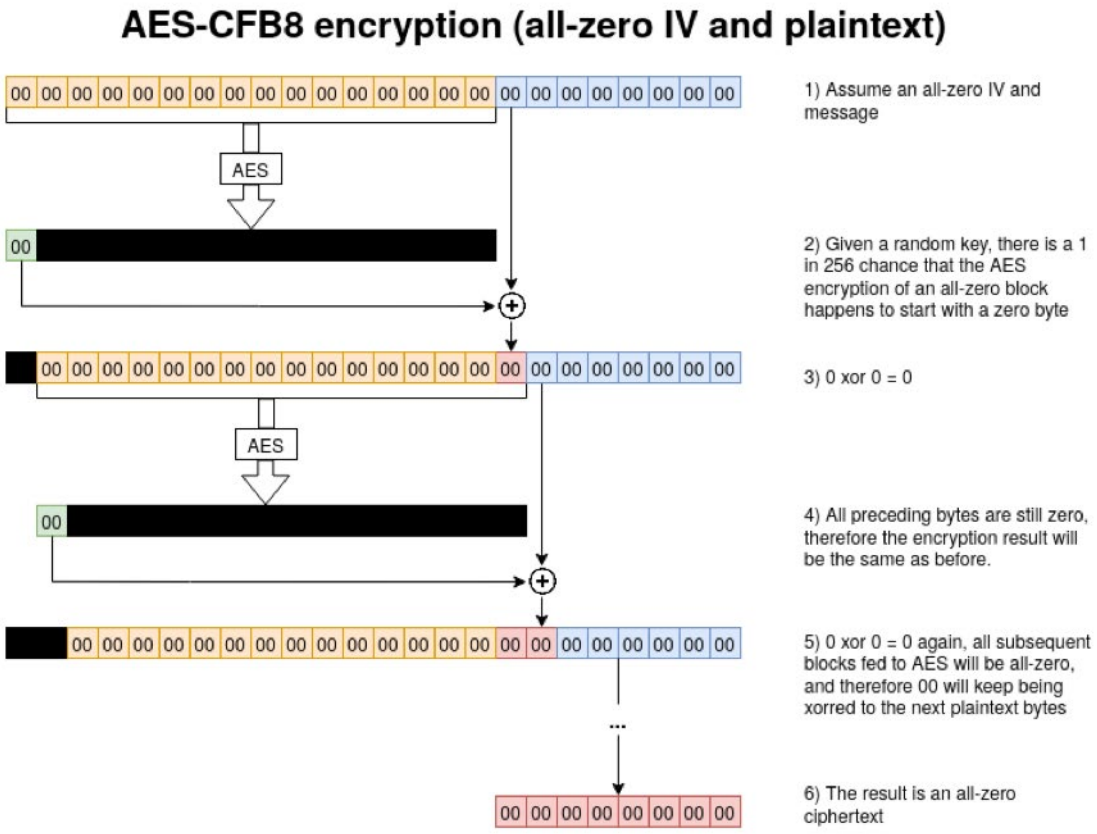

Figure 3: When encrypting a message consisting only of zeroes, with an all-zero IV, there is a 1 in 256 chance that the output will only contain zeroes as well.

In fact, this property is a bit more general: when an IV consists of only zeroes, there will be one integer $0 \leq X \leq 255$ for which it holds that a plaintext that starts with n bytes with value X will have a ciphertext that starts with n bytes with value 0. X depends on the encryption key and is randomly distributed.

In order to attack Netlogon, we don't need the more general property: it is enough to know that an all-zero input can result in an all-zero output. So let's see how we can exploit this.

Exploit step 1: spoofing the client credential

After exchanging challenges with a *NetrServerReqChallenge* call, a client then authenticates itself by doing a *NetrServerAuthenticate3* call. This call has a parameter called *ClientCredential*, and it is computed by applying the *ComputeNetlogonCredential* to the client challenge that was sent in the previous call. Since this challenge can actually be chosen arbitrarily by us, there's nothing stopping us from setting this challenge to 8 zeroes. This means that for 1 in 256 session keys, the correct *ClientCredential* will also consist of 8 zeroes!

So how do we know our session uses one of these keys? Well, we don't. But every time we try to authenticate like this the server will still be generating a unique server challenge that will also be a parameter of the session key derivation. This means that the session key will be different (and uniformly distributed) for every authentication attempt. Since computer accounts are not locked after invalid login attempts, we can simply try a bunch of times until we hit such a key and authentication succeeds. The expected average number of tries needed will be 256, which only takes about three seconds in practice.

With this method, we can log in as any computer in the domain. This includes backup domain controllers, and even the targeted domain controller itself!

Exploit step 2: disabling signing and sealing

While step 1 allows us to bypass the authentication call, we still have no idea what the value of the session key is. This becomes problematic due to Netlogon's transport encryption mechanism ("RPC signing and sealing"), which uses this key but a completely different scheme than the vulnerable *ComputeNetlogonCredential* function.

Luckily, for us, signing and sealing is optional, and can be disabled by simply not setting a flag in the *NetrServerAuthenticate3* call. Modern clients will by default refuse to connect when this flag is not set by the server (likely a measure to prevent downgrade attacks), but servers will not refuse clients that request no encryption. I assume this might be a design choice to maintain legacy compatibility.

Since we act as the client during this attack, we can simply omit the flag and continue.

Exploit step 3: spoofing a call

Even when call encryption is disabled, every call that does something interesting must contain a so-called *authenticator* value. This value is computed by applying *ComputeNetlogonCredential* (with the session key) to the value *ClientStoredCredential* + *Timestamp*.

ClientStoredCredential is an incrementing value maintained by the client. When performing the handshake, it is intialised to the same value as the *ClientCredential* we provided. This client credential consists solely of zeroes, so *ClientStoredCredential* will be 0 for the first call performed after authentication.

Timestamp should contain the current Posix time, and is included in the call by the client along with the authenticator. It turns out, however, that the server does not actually place many restrictions on what this value can be (which makes sense, otherwise clock skew would become very troublesome), so we can simply pretend that it's January 1st, 1970 and also set this value to 0.

If we got through step 1, we also know that *ComputeNetlogonCredential*(0) = 0. So we can authenticate our first call by simply providing an all-zero authenticator and an all-zero timestamp.

Exploit step 4: changing a computer's AD password

So now that we can send a Netlogon call as any computer, what shall we do? There are a number of calls related to account database replication, but these have been disabled since the introduction of Active Directory, so unfortunately we can't use them to extract credentials.

Another interesting call is *NetrServerPasswordGet*, which allows getting an NTLM hash of a computer password. Unfortunately this hash is encrypted with the session key, using yet another mechanism, so this is not useful for us.

What we can exploit, however, is the *NetrServerPasswordSet2* call. This is used to set a new computer password for the client. This password is not hashed but it is encrypted with the session key. How? Well, again using CFB8 with an all-zero IV!

The plaintext password structure in the Netlogon protocol consists of 516 bytes. The final four bytes indicate the password length in bytes. All bytes in the structure that are not part of the password function are seen as padding and can have arbitrary values.

If we simply provide 516 zeroes here, this will be decrypted to 516 zeroes, i.e. a zero-length password. It turns out that setting empty passwords for a computer is not forbidden at all, so this means we can set an empty password for any computer in the domain! (see Figure 4.)

Once that is done, we can set up a new Netlogon connection on behalf of this computer. This time we know the computer's password (it's empty), so we can follow the protocol normally. If we wish, we can now set any other non-empty password as well.

When changing a computer password in this way it is only changed in the AD. The targeted system itself will still locally store its original password. That computer will then not be able to authenticate to the domain anymore, and it can only be re-synchronized through manual action. So at this point we already have a pretty dangerous denial-of-service exploit that allows us to lock out any device from the domain. Also, whenever a computer account has special privileges within a domain, these can now be abused.

Exploit step 5: from password change to domain admin

One of the computers of which we can change the password is that of the domain controller itself, even when this is the same domain controller we are connecting to over Netlogon. Doing so creates an interesting situation, where the DC password stored in AD is different from the password stored in its local registry (at *HKLM\SECURITY\Policy\Secrets\$machine.ACC*). This appears to cause the DC to misbehave in various unpredictable ways (in my lab setup, its DNS resolver stopped working for example).

As an attacker, we would like to use this to log in to the DC using its own password, so that we can compromise it.

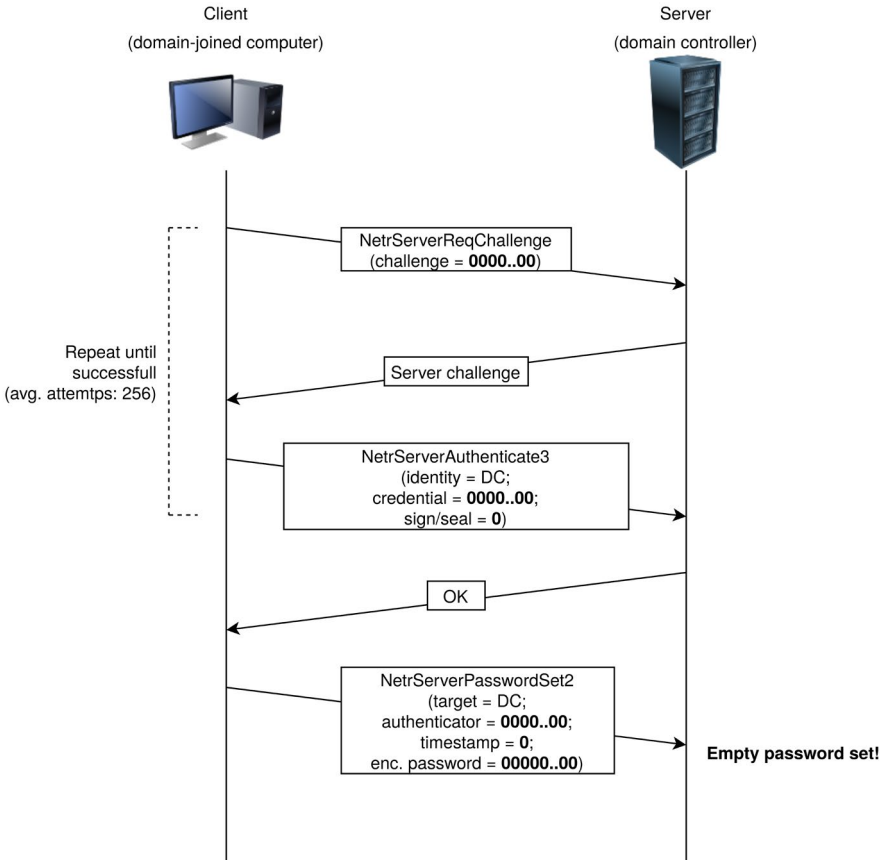

Figure 4: The Zerologon attack, which effectively boils down to filling particular message parameters with zeroes and retrying the handshake a few times in order to set an empty computer password on the DC.

However, this only works when the DC uses the password stored in AD to validate our login attempt, rather than the one stored locally. After some experimentation, I found that simply running Impacket's '*secretsdump*' script with the new DC password worked. This script will successfully extract all user hashes from the domain through the Domain Replication Service (DRS) protocol. This includes domain administrator hashes (including the '*krbtgt*' key, which can used to create golden tickets), that could then be used to login to the DC (using a standard pass-the-hash attack) and update the computer password stored in the DC's local registry. Now the DC behaves normally again, and the attacker has become domain admin.

Conclusion

By simply sending a number of Netlogon messages in which various fields are filled with zeroes, an attacker can change the computer password of the domain controller that is stored in the AD. This can then be used to obtain domain admin credentials and then restore the original DC password.

This attack has a huge impact: it basically allows any attacker on the local network (such as a malicious insider or someone who simply plugged in a device to an on-premise network port) to completely compromise the Windows domain. The attack is completely unauthenticated: the attacker does not need any user credentials.

The patch released on Patch Tuesday of August 2020 addresses this problem by enforcing *Secure NRPC* (i.e. Netlogon signing and sealing) for all Windows servers and clients in the domain, breaking exploit step 2. Furthermore, my experiments show that step 1 is also blocked, even when not dropping the sign/seal flag. I don't know how exactly this is implemented: possibly by blocking authentication attempts where a *ClientCredential* field starts with too many zeroes. I did not succeed in bypassing this check. Either way, the Zerologon attack such as described here will no longer work if the patch is installed.

If practical ways would exist to bypass the step 1 protections (perhaps involving a lot of additional brute-forcing), this could put legacy or third-party devices at risk for which Secure NRPC is not enforced. An attacker could then still reset the computer password of these devices as stored in AD, which would deny service by effectively disconnecting those devices from the domain. Potentially this would also allow man-in-the-middle attacks similar to **CVE-2019-1424**, with which an attacker could get local admin access to these particular devices.

To address this remaining risk, Windows will log warning events when such devices exist in the domain. The option also exists to turn on "enforcement mode" which will mandate Secure NRPC for all devices, even when this would cause them to break. In February 2021 this enforcement mode will be turned on by default, requiring administrators to update, decommission or whitelist devices that do not support Secure NRPC beforehand. See **Microsoft's guide** for more information.