# Exploiting known vulnerabilities, misconfigurations and weaknesses in native protocols to compromise Windows Active Directory Domains with a focus on traceability and ease of use.

Søren Fritzbøger - s153753
Vejledt af Henrik Tange

20. Januar 2019

Danmarks Tekniske Universitet

## Abstract

Abstract here

# Table of contents

## Abbreviations

**AD** Active Directory. 22

**AV** Antivirus. 5, 24

**DC** Domain Controller. 6, 23

**GPU** Graphics Processing Unit. 15

**HTTP** HyperText Transfer Protocol. 14, 15, 19–21, 26

**IDS** Intrusion Detection System. 5

**KDC** Key Distribution Center. 23

**LLMNR** Link-local Multicast Name resolution. 5–7, 11–14, 21, 26

**LSA** Local Security Authority. 25

**LSASS** Local Security Authority Subsystem Service. 21–24

**NBNS** NetBIOS Name Resolution. 5–9, 11, 14, 21, 26

**PTH** Pass-the-hash. 22, 23

**RDP** Remote Desktop Protocol. 21, 22

**SIEM** Security Information and Event Management. 5

**SMB** Server Message Block. 14, 15, 17–19, 21, 23, 26

**SSO** Single Sign-On. 14, 21

**SSP** Security Support Provider. 14, 17, 19, 21

**SSPI** Security Support Provider Interface. 14

**winrs** Windows remote Management. 22

**WINS** Windows Internet Name Service. 8

**WMI** Windows Management Instrumentation. 22, 23

# 1 Introduction

## 1.1 Problem background

When performing a pentest on Windows Active Directory environments, one of the goals is usually to obtain Domain Administrator privileges, usually in the form of a Domain Admin account. There are numerous ways to gain initial foothold in an Active Directory Domain, but the most common one is by exploiting the native protocols NetBIOS Name Resolution (NBNS) and Link-local Multicast Name resolution (LLMNR)[24] to gain crackable hashes which can be bruteforced offline. Hereafter a number of different techniques and exploits can be used to gain additional credentials, but it usually consists of dumping cached credentials on domain joined hosts in one way or another.

As one can easily figure out this is usually a manual job where many different tools are joined together to produce the right result. This usually means it is a trivial and easy job, which requires a lot of time that can be spent on more advanced tasks. This is often done with tools not written by the pentester themselves, which can pose a security risk as the tools can be backdoored or otherwise have security vulnerabilities.

One of the objectives of a pentest is usually to be as silent as possible and not trigger any alerts in any Intrusion Detection System (IDS), Security Information and Event Management (SIEM) or similar systems. The risk of using publicly available tools is therefore also that the tools are highly likely to be detected by Antivirus (AV) and will often trigger unwanted alerts.

Another issue of performing pentests is to document and remember the order of tasks that where done. A pentest usually concludes with a report where the necessary steps are explained to the customer, and this includes in what order tasks were done and which user credentials were used.

## 1.2 Problem brief

The purpose of this project is to determine whether a proper solution to the aforementioned problem can be found, and to analyze how such a tool can be developed. The problem is split in two, where the first goal is to gain an initial foothold and the second is to gain Domain Administrator privileges. To accomplish this the many techniques and methods will be discussed and evaluated in comparison to each other, and the most valid solution will be implemented in a piece of software that aims to be easy to use and contain a high level of traceability.

The developed piece of software must be able to be easy to use so that an incentive to use it instead of other tools is created. It should be constructed with AV evasion in mind, such that it will not be detected by AVs. Furthermore it must be designed with traceability in mind, such that a clear timeline can be constructed and documented.

### 1.3 Report structure

### 1.4 Pentesting Windows Domains

## 2 Initial foothold

In a Windows Active Directory Domain there are numerous ways of gaining an initial foothold. The following methods are the most used in modern penetration testing of Windows AD environments.

**BRUTE** User credential bruteforcing

**SPRAY** Password spraying

**EXPL** Exploiting known vulnerabilities on unpatched systems

**CLEAR** Clear text passwords stored on public shares

**SPOOF** NBNS and/or LLMNR spoofing

All of the above mentioned methods have their weaknesses and strengths, which should be taken into account when choosing the best method or methods to gain initial foothold in the domain. To make an educated guess of which method(s) to pursue further a comparison between the different methods is needed. Table 1 gives a comparison of the different methods, and shows which weaknesses and strengths each methods possess.

| **Strength** | **BRUTE** | **SPRAY** | **EXPL** | **CLEAR** | **SPOOF** |
|---|---|---|---|---|---|
| **Is it automatable?** | + | + | - | - | + |
| **Is it fast?** | - | - | + | - | - |
| **Account lockout issues?[12]** | - | - | + | + | + |
| **Communication with critical systems such as a DC?** | - | - | + | + | + |
| **Easy to detect?** | - | - | + | + | + |
| **Is it easy to do?** | + | + | - | (+)[1] | + |
| **Points** | 2 | 2 | 4 | 3.5 | 5 |

Table 1: Comparison of different methods to gain initial foothold in a Windows AD environment

All of the above mentioned methods are valid and have been used during real penetration tests. Table 1 scores each method according to their pros and cons, and here it is clearly showed that spoofing NBNS and LLMNR is the most optimal way of gaining initial foothold. This corresponds with real life experience where the protocols are enabled by default[21] and not monitored

[1] This method can be very time consuming

correctly.
Now that a method has been chosen, section 2.1 will look further into how spoofing can be done in an automated way.

## 2.1 Spoofing

To understand how spoofing of LLMNR and NBNS works we first need to elaborate how spoofing can lead to a credential compromise. To do this we need to understand how name resolution works in Windows, regardless of the protocol used. Windows follows a sequence of steps in order to resolve a host name.[15] The steps are the following:

1. The client checks to see if the name queried is its own
2. The client searches local Hosts file
3. The client queries the DNS server
4. If enabled, Name resolution is done (LLMNR and NBNS)

This is illustrated on figure 1 where it is also illustrated how spoofing fits into the sequence. As it is shown, the Attacker will listen to multicast packets sent on the local subnet and answer to any Name Resolution packets. If successful, the *Client* will register *Server1* to have the IP address of *Attacker*, and thereby sending all packets intended for *Server1* to *Attacker*.

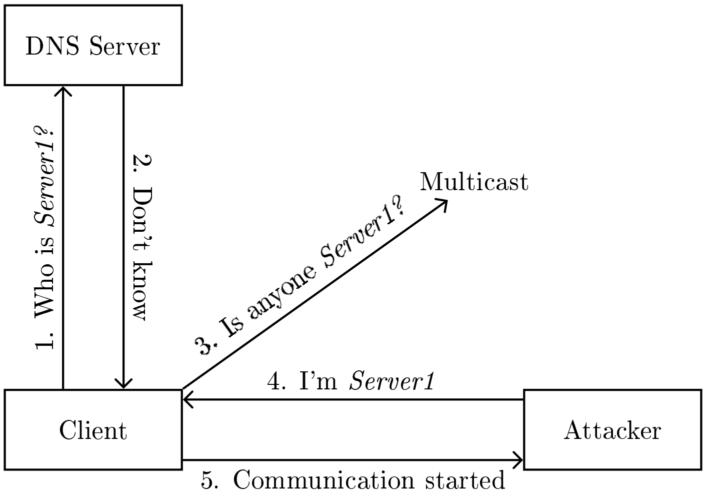


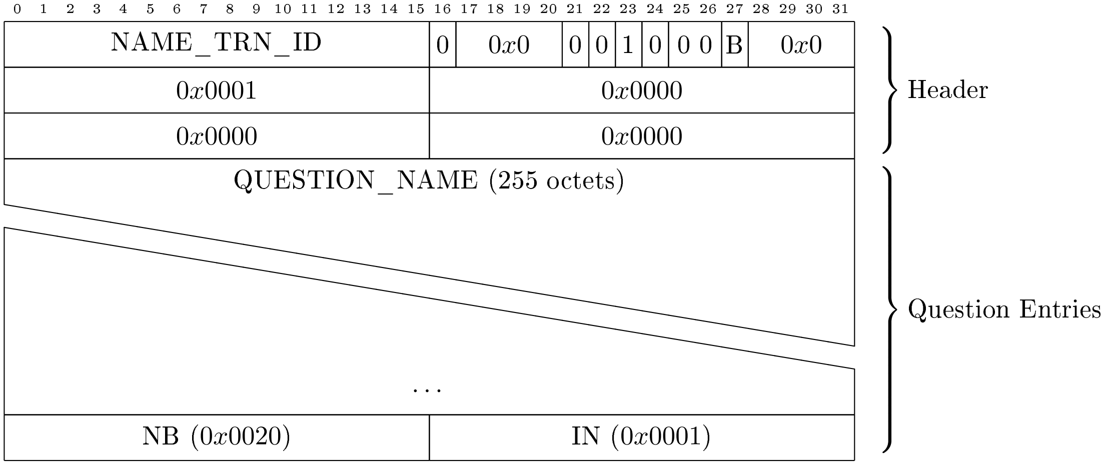
Figure 1: How an attacker spoofs the Name Resolution protocols in order to receive traffic intended for other servers

In Windows two different Name Resolution protocols are used and operate by default on all machines. LLMNR and NBNS work side by side unless specifically

turned off, which it is not by default. In section 2.1.2 and 2.1.2 the protocols are analyzed in detail and the attacks are described in detail.
After successfully spoofing a host name and getting traffic redirected to our machine, we need to be able to use that traffic for a malicious purpose. The most common purpose is to gather credentials from the client by having rouge servers running on the attacker. The technique and methods behind this is explained in section 2.2

### 2.1.1 NetBIOS Name Resolution (NBNS)

In Windows NBNS is implemented in the Windows Internet Name Service (WINS) which is a legacy service used to map host names to IP addresses. In newer versions of Windows it has no use, but it is kept for backward compatibility purposes. The NetBIOS RFC specification, RFC 1001[7], contains much more than Name Resolution, but for spoofing purposes we only need to look at Name Resolution. RFC 1002[8] contains detailed technical specification as to how Name Resolution is implemented in NetBIOS.
NBNS has many other features such as Name Registration, Name Refresh etc. but in order to spoof name resolution we need to understand *Name Query Request* packets and respond with *Name Query Response* packets. The format of a *Name Query Request* is specified on figure 2. As it can be seen, only the fields **NAME_TRN_ID** and **QUESTION_NAME** are necessary to read in order to generate a valid *Name Query Request*.



Where

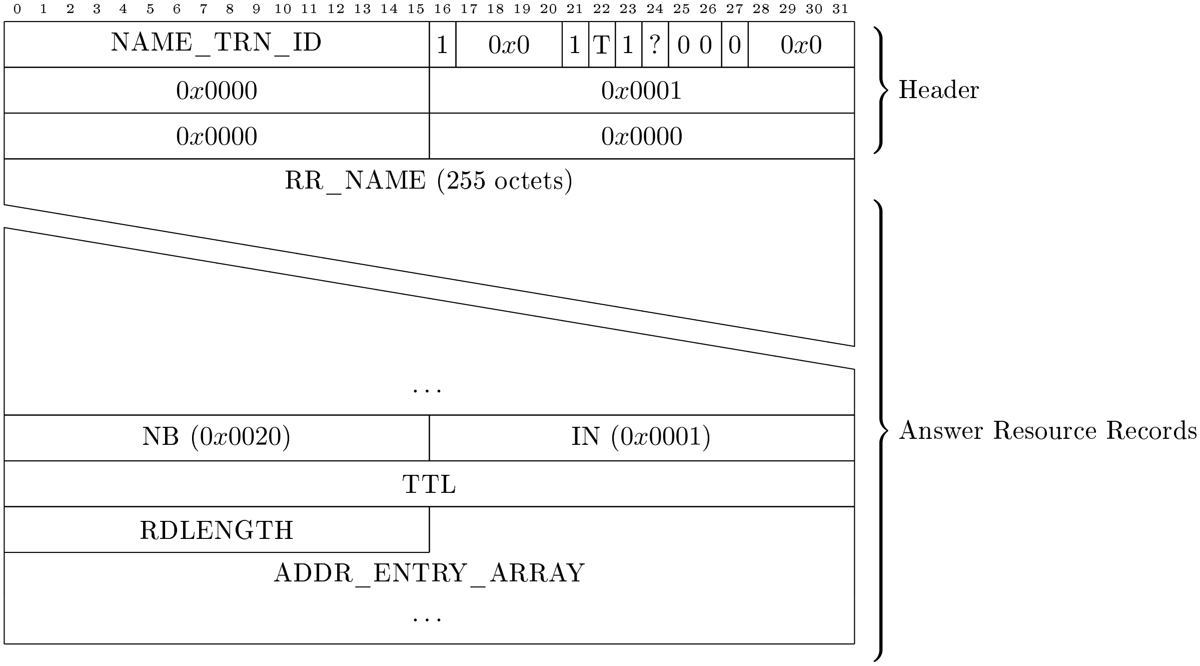
**NAME_TRN_ID** is the transaction ID for the Name Service Transaction

**QUESTION_NAME** is the compressed name of the NetBIOS name for the request.

Figure 2: NetBIOS Name Resolution (NBNS) Name Query Request[8, sec. 4.2.12]

To send a spoofed response to the NBNS request we need to reply with a *Name Query Response*. The structure of a response packet can be seen in figure 3. It is important to mention that question name is encoded using a very mysterious encoding and truncated to a maximum of 16 bytes. Each half-byte of the characters ASCII code is added to the ASCII Code 'A'[25], which result in every byte occupying two bytes in the resulting packet. So SERVER1[2] will be encoded to FDEFFCFGEFFCDBCACACACACACACACACA where CA is empty padding until the length is 32 bytes. Figure 3 shows the structure of a response, and describes the values of the different fields.

[2] NetBIOS names are always uppercase



| NAME_TRN_ID | 1 | 0$x$0 | 1 | T | 1 | ? | 0 0 | 0 | 0$x$0 |
|---|---|---|---|---|---|---|---|---|---|
| 0$x$0000 | 0$x$0001 | | | | | | | | |
| 0$x$0000 | 0$x$0000 | | | | | | | | |

RR_NAME (255 octets)

...

| NB (0$x$0020) | IN (0$x$0001) |
|---|---|

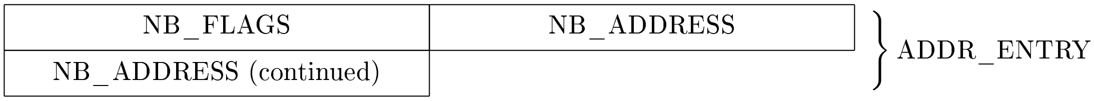
TTL

RDLENGTH

ADDR_ENTRY_ARRAY

...

Where each ADDR_ENTRY has the following format:



| NB_FLAGS | NB_ADDRESS |
|---|---|
| NB_ADDRESS (continued) | |

And

**NAME_TRN_ID** is the transaction ID from the request

**RR_NAME** is the question name from the request

**TTL** is the time to live for the response in seconds

**RDLENGTH** is the length of the data field (ADDR_ENTRY_ARRAY)

**ADDR_ENTRY_ARRAY** is zero or more of the following:

> **NB_FLAGS** consists of three different things. Bit 0 is Group Name Flag (0), bit 1-2 is Owner Node Type (00) and bit 3-15 is reserved for future use (all 0)
>
> **NB_ADDRESS** is an IP Address. In this case our own IP Address

Figure 3: NetBIOS Name Resolution (NBNS) Name Query Response[8, sec. 4.2.13]

Figure 4 shows a real request sent from a Windows Server 2016 requesting the host for **Server1**. The corresponding values of **NAME_TRN_ID** and **RR_NAME** is $0xd056$ and FDEFFCFGEFFCDBCACACACACACACACACACA respectively. So as figure 3 shows, we need to construct a packet with the specified transaction ID and host name. Additionally, we set **TTL** to $0x000000a5$ (2 minutes and 45 seconds), **RDLENGTH** to 6 ($0x0006$) and NB_ADDRESS to our own IP Address 10.211.55.4 ($0x0ad33704$). This results in the packet seen on figure 5 which is a direct response to 4 leading to a successful spoof.

```
0000   d0 56 01 10 00 01 00 00 00 00 00 00 20 46 44 45   ÐV.......... FDE
0010   46 46 43 46 47 45 46 46 43 44 42 43 41 43 41 43   FFCFGEFFCDBCACAC
0020   41 43 41 43 41 43 41 43 41 43 41 43 41 00 00 20   ACACACACACACA..
0030   00 01                                             ..
```

Figure 4: NBNS Name Query Request

```
0000   00 1c 42 83 2d 74 00 1c 42 1b 5b 6c 08 00 45 00   ..B.-t..B.[l..E.
0010   00 5a 56 d5 00 00 80 11 00 00 0a d3 37 04 0a d3   .ZVÕ.......Ó7..Ó
0020   37 03 00 89 00 89 00 46 84 04 d0 56 85 00 00 00   7......F..ÐV....
0030   00 01 00 00 00 00 20 46 44 45 46 46 43 46 47 45   ...... FDEFFCFGE
0040   46 46 43 44 42 43 41 43 41 43 41 43 41 43 41 43   FFCDBCACACACACAC
0050   41 43 41 43 41 43 41 00 00 20 00 01 00 00 00 a5   ACACACA.. .....¥
0060   00 06 00 00 0a d3 37 04                           .....Ó7.
```

Figure 5: NBNS Name Query Response

### 2.1.2 Link-local Multicast Name resolution (LLMNR)

LLMNR is the newest protocol for name resolution where DNS name resolution is not possible[1]. LLMNR works in the same way as NBNS in such that a name query is sent to the link-scope multicast address(es), and a responder can hereafter respond to the packet and claim itself as the host that was requested. The sequence of events with LLMNR according to RFC 4795[1] is the following

1. An LLMNR sender sends a LLMNR query to the link-scope multicast address(es) on port 5355. This is a LLMNR packet containing a Question Section
2. A responder responds to this query by sending an UDP packet. This is a LLMNR packet containing a Question Section and a Resource Record
3. The sender process the responders packet

This is all well if we assume that the network itself is not already compromised. In case a malicious host is existent on the network, and the host is listening on the link-scope multicast address there is nothing from stopping this host in responding maliciously to the packets. LLMNR is made to follow the DNS specification, so in order to spoof it we need to know how DNS packets look like. This can be found in RFC 1035[22].

**LLMNR packets** There exists two different LLMNR packet types. A **request** and a **response**. The **request** contains the *header* and a *question section*. The **response** contains a *header*, a *question section* and a *resource record*. Format details of these types can be seen in figure 6.

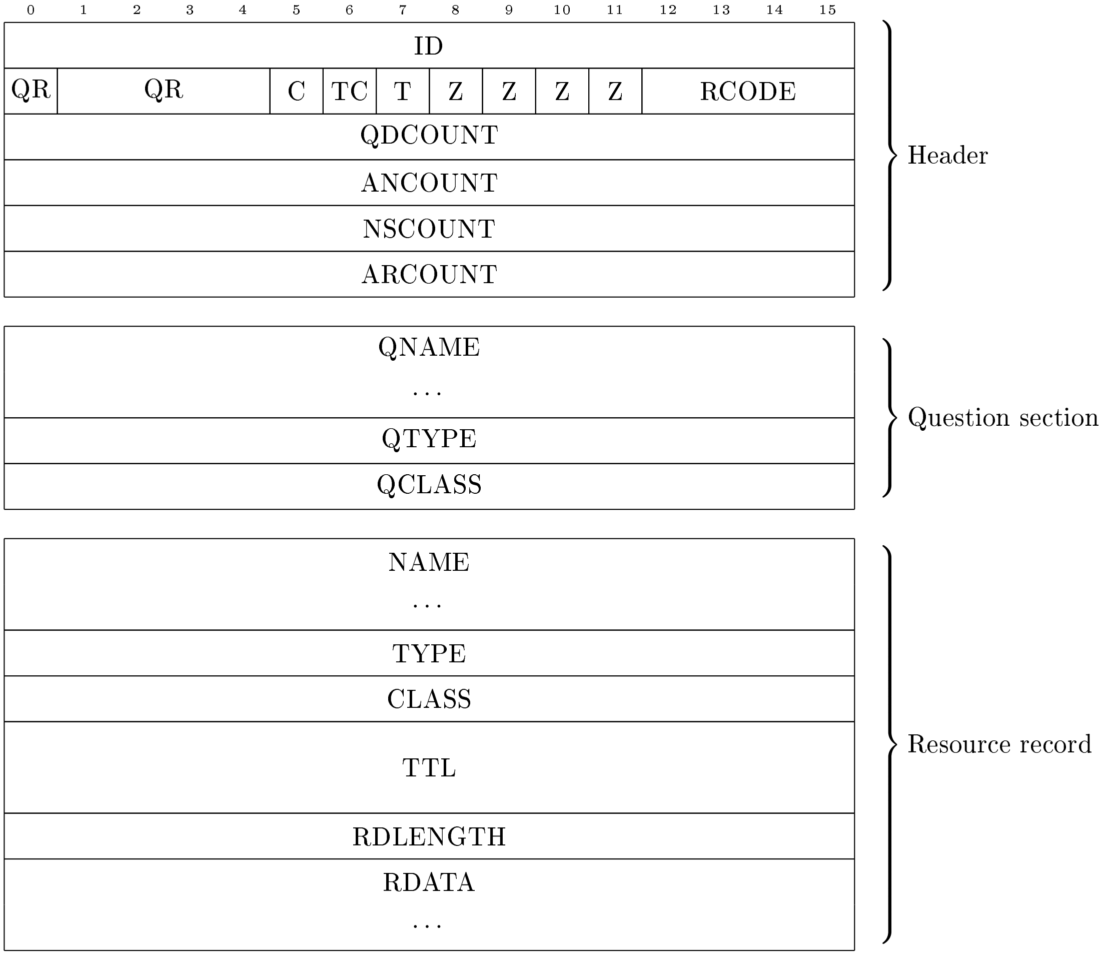


Figure 6: Link-local Multicast Name resolution (LLMNR) packet specification[1][22]

Where

**QNAME** is a domain name in the following format: A length octet followed by that number of octets

**QTYPE** is a two octet code which specify the query type. Usually $0x0001$ for Host address(IP)

**QCLASS** is a two octet code which specify the query class. Usually $0x0001$ for Internet (IN)

And

**NAME** See QNAME of *Question Section*

**Type** See TYPE of *Question Section*

**CLASS** See QCLASS of *Question Section*

**TTL** is a 32 bit unsigned integer which specify Time To Live in minutes

**RDLENGTH** is a 32 bit unsigned integer which specify the number of octets in RDATA

**RDATA** is a variable length string of octets. Usually an IP address

Figure 6: Link-local Multicast Name resolution (LLMNR) packet specification[1][22]

This report will not explain the packet details in full, but will focus on the parts necessary to spoof a LLMNR response. The most important fields of the Question Section and the Resource Record is explained in the figure.
So to answer a LLMNR packet we need to create a *Resource Record* to match the *Question section* sent out by a client. *Name, Type and Class* should match the request, *TTL* should be set to an arbitrary time in minutes (for example 30 - $0x0000001e$ in bytes), *RDLENGTH* should be 4 ($0x0004$) and RDATA should be our own IP Address.

```
0000   f2 75 00 00 00 01 00 00 00 00 00 00 07 73 65 72  .u...........ser
0010   76 65 72 31 00 00 01 00 01                       ver1.....
```

Figure 7: LLMNR request

```
0000   f2 75 80 00 00 01 00 01 00 00 00 00 07 73 65 72  .u...........ser
0010   76 65 72 31 00 00 01 00 01 07 73 65 72 76 65 72  ver1......server
0020   31 00 00 01 00 01 00 00 00 1e 00 04 c0 a8 00 02  1...........@.b
```

Figure 8: Spoofed LLMNR response

Figure 7 and 8 shows how the response should look for a request for *server1*.

After sending this response we would have succeeded in spoofing the LLMNR protocol to send traffic to our host.

## 2.2 Credential acquiring

After successfully spoofing either a NBNS or LLMNR request, we have now successfully imposed as another host meaning all traffic intended for that host will be directed to us. It is also important to mention that name resolution of non-existing hosts will still be spoofed and therefore be registered by the sender of the request as belonging to us. In Windows, name resolution often happens when trying to request either a Server Message Block (SMB) share or a website using HyperText Transfer Protocol (HTTP). Luckily for us Windows has implemented the Security Support Provider Interface (SSPI) which allows an application to use various security models available on a computer, which works as a Single Sign-On (SSO) solution that allows users to easily authenticate to various services using their cached credentials[19]. We can use this to our advantage by implementing a Security Support Provider (SSP) by creating fake SMB and HTTP services. There exists a couple of different SSP's including Negotiate, NTLM, Kerberos, Digest SSP and others. Microsoft recommends that you use Negotiate as it acts as an application layer between the SSPI and the different SSP's usually choosing Kerberos over NTLM as it is more secure. Though, we can force our service to use the NTLM SSP, which will give us a hash that we can crack offline. The different types of hashes will be discussed in more detail in section 2.2.1.

### 2.2.1 Credential types

In the windows ecosystem there is a lot on confusion on the different types of hashes and where they are used. This short section aims to describe the different hashes briefly to avoid confusion later on in the project. There exists 4 (or 5 depending on who you ask) different hashes in windows[6]

**LM** LM is the original hash type used by Windows dating back to OS/2. LM has a number of shortcomings, but the biggest is that it is a maximum length of 14 characters, which is then split into two 7-character chunks, where each part is then encrypted with the string "KGS#$%" and concatenated. The obvious flaw here is that you only have to crack two 7-character hashes instead of one, which can easily be done with modern GPUs in mere hours.

**NT** NT is the current Windows standard for hashing passwords. This is basically just a MD4 of the little endian UTF-16 of the password. MD4 has its obvious flaws concerning collision attacks, but such attacks and methods are out of scope for this project.

**NTLM** NTLM is a combination of LM and NT hashes in the form *LM:NT*. This hash type can be used in attacks known as pass-the-hash[16] where you can authenticate using the NTLM password instead of the clear-text password. So having the NTLM hash is in most attack scenarios essentially the same as having the clear-text password.

**NetNTLMv1** NetNTLMv1 is Microsofts first attempt at a challenge/response hash between a client and a server. It will use either the NT or LM hash to generate the NetNTLMv1 hash. Once again this uses DES and has obvious flaws allowing you to convert it to three different DES keys which can be cracked with much less computer power and converted into an NTLM hash[5].

**NetNTLMv2** NetNTLMv2 is the newest challenge/response based hash used for network authentication. This hash uses a 8-byte server and client challenge combined with the current time and domain name to create a more secure hash. It also uses HMAC-MD5 as algorithm, which is more secure than DES. NetNTLMv2 will also use either the NT or LM hash to generate the hash.

### 2.2.2 NTLM

NTLM is the primary Challenge/Response protocol used in Windows authentication, and can easily be encapsulated in other protocols such as SMB and HTTP. The NTLM authentication protocol consists of three message types and is therefore a simple protocol[14]. The three message types are the following:

**Negotiate** The client initiates the authentication.

**Challenge** The servers sends a 16 byte challenge to the client

**Authenticate** The client encrypts the challenge with the user's hash and sends it to the server.

This is a very simple authentication protocol which has it's obvious flaws. Once you've gotten the encrypted challenge back bruteforcing the password is very trivial. The encrypted challenge returned is either a NetNTLMv1 or NetNTLMv2 hash, as described in section 2.2.1, and can either be converted to a passable hash or bruteforced quickly using a couple of modern Graphics Processing Unit (GPU)'s. All NTLM messages start with the protocol identifier *NTLMSSP + a null byte (0x00)*[23]. After the identifier a type byte (0x01 for negotiate, 0x02 for challenge and 0x03 for authenticate), three null bytes, a four byte flag and another two null bytes. As mentioned, the server sends a **Challenge** message to the client (detailed on figure 9), whereafter the client replies with an **Authenticate** message (detailed on figure 10).

**NTLM Challenge Message** The Challenge message contains a number of predefined fields such as protocol, flags and zero bytes, and the only not-predefined field is the eight byte challenge that is encrypted with the users password. The challenge is chosen by the server and should, according to the specification, change with every request. But seeing as we are implementing the server ourself, we can choose our own challenge and keep it the same for every request. This is particularly useful for when the password needs to be cracked as we can create rainbow tables beforehand.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PROTOCOL | | | | | | | |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| msg len | | 0 | 0 | FLAGS | | 0 | 0 |
| CHALLENGE | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Where

**PROTOCOL** is always *NTLMSSP* followed by a null-byte

**CHALLENGE** 8 arbitrary bytes chosen by the server

**FLAGS** always set to $0x8201$

Figure 9: NTLM Challenge message

**NTLM Authenticate message** The Authenticate message consists of five parts, namely Domain, User, Host, LM and NT. Each part has a length field, which for unknown reasons are duplicated in the protocol, a offset part and the actual content. Each part is separated by two null bits. This can be seen in detail on figure 10. The contents of this message can be combined into an NetNTLMv2 hash with the format *User::Domain:Challenge:LM:NT*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PROTOCOL | | | | | | | |
| 3 | 0 | 0 | 0 | LM-LEN | | LM-LEN | |
| LM-OFF | | 0 | 0 | NT-LEN | | NT-LEN | |
| NT-OFF | | 0 | 0 | DOMAIN-LEN | | DOMAIN-LEN | |
| DOMAIN-OFF | | 0 | 0 | USER-LEN | | USER-LEN | |
| USER-OFF | | 0 | 0 | HOST-LEN | | HOST-LEN | |
| HOST-OFF | | 0 | 0 | 0 | 0 | 0 | 0 |
| MSG-LEN | | 0 | 0 | $x$01 | $x$82 | 0 | 0 |
| DOMAIN<br>. . . | | | | | | | |
| USER<br>. . . | | | | | | | |
| HOST<br>. . . | | | | | | | |
| LM<br>. . . | | | | | | | |
| NT<br>. . . | | | | | | | |

Where

**PROTOCOL** is always *NTLMSSP* followed by a null-byte

**CHALLENGE** 8 arbitrary bytes chosen by the server

Figure 10: NTLM Authenticate message

### 2.2.3 Server Message Block (SMB)

As stated in section 2.2 we need to implement a NTLM SSP in SMB. In SMB this is done by encapsulating the NTLM authentication into SMB. A SMB session is first started whereafter NTLM authentication happens and then the SMB session is continued. For this project we do not need to implement the full SMB protocol, as we are only interested in the NTLM authentication and the encrypted challenge we get hereof. The full flow of this process can be seen in figure 11 which starts after a client has successfully been spoofed.

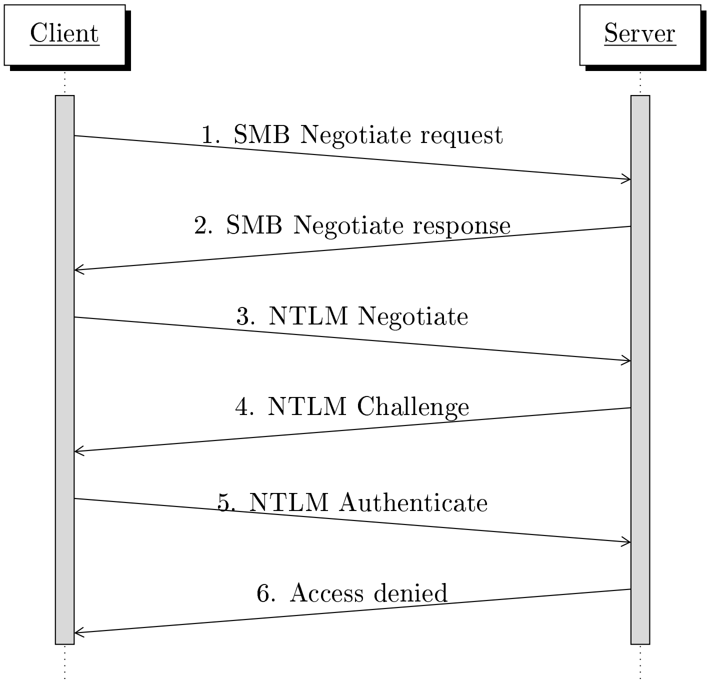


Figure 11: SMB NTLM authentication[17]

The authentication flow of SMB, which can be seen on figure 11, uses the following messages in the mentioned order

**1. SMB_COM_NEGOTIATE** The client first sends a SMB_COM_NEGOTIATE message whose purpose is to negotiate the supported authentication types.

**2. SMB_COM_RESPONSE** The server responds with a SMB_COM_REPONSE stating the used authentication method that both the client and server supported. In our case this will always be set to NTLM authentication as we are not interested in Kerberos authentication.

**3. SMB_COM_SESSION_SETUP_ANDX request 1** The client sends an encapsulated NTLM Negotiate message to the server.

**4. SMB_COM_SESSION_SETUP_ANDX response 1** The server receives the NTLM negotiate message and replies with a NTLM Challenge

**5. SMB_COM_SESSION_SETUP_ANDX request 2** The client sends a NTLM Authenticate message containing the encrypted challenge

**6. Access denied** An access denied response is sent to close the connection to the client. At this point we will have received a NetNTLMv1 or NetNTLMv2 hash, so we don't wish to continue the session.

In step **(5)** we receive an *NTLM Authenticate* message which contains the encrypted challenge. An example of such a message can be seen on figure 12. Using the data given in this message, we can construct a NetNTLMv2 password hash which can be cracked offline using various cracking techniques such as bruteforcing or rainbow tables.

Figure 12: SMB NTLM Authenticate Message

### 2.2.4 HyperText Transfer Protocol (HTTP)

Implementing a NTLM HTTP SSP is, once again, encapsulating the NTLM authentication messages into HTTP. This is done using the headers **WWW-Authenticate** and **Authorization** for the server and client respectively. The flow is very similar to that of SMB, and can be seen on figure 13. The flow starts with a client accessing a server on the standard http port 80[3], with the server responding with a 401 Unauthorized and supplying the header *WWW-Authenticate: NTLM* to let the client know that the server supports NTLM authentication. After this the standard NTLM authentication occurs with Negotiate, Challenge and Authenticate messages encapsulated in HTTP. One important point to highlight is that the NTLM messages are encoded using base64 as HTTP is a text based protocol. After a successful NTLM Authentication over HTTP we are once again left with a Authenticate message containing the necessary information to construct a NetNTLMv2 password hash that can be cracked offline using various techniques.

[3] In theory this can be any arbitrary port but seeing as we're obtaining data based on spoofing we need to listen on port 80

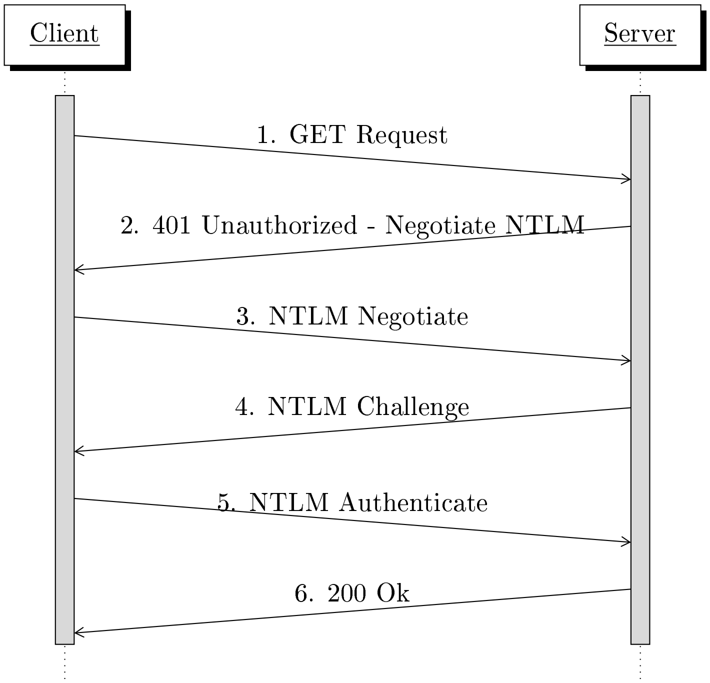


Figure 13: HTTP NTLM authentication[23]

**1. GET Request** The client initiates the connection with a GET request to the server

**2. 401 Unauthorized** The server responds the GET requests with a 401 Unauthorized response containing the HTTP header *WWW-Authenticate: NTLM* letting the client know that it should authenticate using NTLM

**3. NTLM Negotiate** The client responds with another GET request containing the base64 encoded NTLM Negotiate message encapsulated in the Authorization header

**4. NTLM Challenge** The server responds with the base64 encoded NTLM Challenge message containing the challenge the client should encrypt in the WWW-Authentication header

**5. NTLM Authenticate** The client responds with the base64 encoded NTLM Authenticate message

**6. 200 Ok** The server now responds with a 200 Ok which let the client know that the credentials were accepted.

# 3 Attack methods

After successfully spoofing a victim by abusing LLMNR and NBNS and getting a NetNTLMv2 hash by running SMB and HTTP SSP's we are at a point where we need to use more traditional methods to compromise the network further. As mentioned in section 2.2.1 both NetNTLMv2 and NetNTLMv1 hashes encrypts the server given challenge with either the NT or LM hash. In other words we can use bruteforcing tools such as hashcat[9] to crack the hashes and gain the original password, which will of course lead to a full user compromise as we then have username and password in clear-text.

In the rare case that a password is sufficiently complex and cannot easily be cracked or otherwise guessed, there still exists methods to utilize spoofing and poisoning to gain unauthorized access to systems. One such method is called NTLM Relaying[2] where you construct your SSP's to relay the credentials to another server/host.[4]

After we have cracked the password we need to move on to compromising the domain further. There are many was of doing this, and it it very dependent on the targets of the pentest. A common target for many Windows pentests is to gain Domain Admin privileges, either by compromising a user already possessing the privilege or by exploiting various vulnerabilities. This project primary goal is to become Domain Admin in a quiet and efficient way, that does not disturb the network and remain as close to undetectable as possible. Using vulnerabilities is first and foremost a very periodic way to achieving Domain Admin privileges, as most known vulnerabilities are fixed quite easily and we are not in the position of owning zero day vulnerabilities which can be easily burned on standard pentests. Luckily there are many other methods to achieve lateral movement in a Windows network, which this project will explore further in the following sections. Analyzing and discussing all the different types of attack would in itself be a very big project, so in this report we will focus on Local Security Authority Subsystem Service (LSASS) that, among other things, handles storage of credentials.

## 3.1 Local Security Authority Subsystem Service (LSASS)

LSASS is a protected subsystem that handles authentication and sessions in Windows. So whenever you login to a Windows machine, access it via Remote Desktop Protocol (RDP) or connect using SMB LSASS will handle the authentication and store the credentials in a safe way afterwards[5]. The feature of storing password in the memory is actually what SSO uses to sign-in automatically to services, and is a very essential part of why poisoning certain protocols work. But, it also gives us other opportunities to steal credentials. After gaining access to one user, using the aforementioned mentioned methods, we are freely

[4] Relaying credentials is outside the scope of this project, but is something that would work well with the rest of setup

[5] LSASS does not always save the credentials in memory, it depends on the type of authentication and how it happens[18]

available to query information in the Active Directory (AD) which can give us information about hosts, user privileges, user groups etc. Using this information it is possible to find hosts where the compromised user has administrator privileges. These privileges can be used to create a remote connection to the host and steal sessions from that particular host. Lets for example say that User1 has administrator privileges on the host jumpserver1. On jumpserver1 three other users are authenticated using RDP and currently have an active session. In this case, he LSASS process (lsass.exe) on the host jumpserver1 will therefore contain a total of four cached credentials in the process memory. The type of password hash stored in memory varies with different Windows systems and level of patching, but a generel rule of thumb is that servers after Windows Server 2008 R1 and clients after Windows 7 mainly saves the password as a NT hash, and hosts before the mentioned versions save it as clear-text[20]. Of course there exists tool to extract these credentials given an interactive login or a memory dump of the LSASS process. This will be discussed in depth in section 3.2.1.

Using this knowledge we clearly see a way of performing lateral movement through the network, and in most cases this will eventually lead to an account with Domain Admin privileges. Although we cannot be sure that the passwords we compromise through LSASS memory dumps are clear-text, this will not pose a problem as explained in section 3.2

## 3.2 Remote access

Remote access should be understood in the sense of getting a interactive session on another host that can be used to interact with this host using either a command line or graphical interface. In Windows there are many official and unofficial ways to achieve this. Among official ways you will find methods such as **Enter-PSSession**, **Invoke-Command**, **PSExec**, **Windows remote Management (winrs)**, **RDP** and **Windows Management Instrumentation (WMI)**. These all work very well when you have valid credentials with clear-text passwords. But as stated in section 3.1 dumping credentials from LSASS memory will in newer Windows versions give you a NT hash and not the clear-text password. In these cases we can utilize a technique called Pass-the-hash (PTH), which will allow us to use NTLM Authentication to authenticate against a remote host using a hashed password.

**Pass-the-hash (PTH)** Pass-the-hash (PTH) attacks is a way of authenticating to a remote host using a NT or LM hash instead of of the clear-text password. If we look closer at the "NTLM Authenticate message" figure from page 17 and the NetNTLMv2 hash from section 2.2.1, we can clearly see that the NetNTLMv2 hash is based on the NT or LM hash and **not** the clear-text password. Even though Windows default to Kerberos authentication, we can still force the host to accept NTLM Authentication unless it is actively disabled. Though, even with Kerberos authentication we can still use the NT or LM hash to create a ticket which can then be passed, but that is outside the scope of this

project and has other implications as it requires direct communication with a Key Distribution Center (KDC) (Usually a DC).
PTH attacks can be used with most Windows protocols such as SMB and WMI, but it requires you to re-implement the protocol to be able to use it, as Windows did not make the feature available in any of their tools. Luckily the Impacket project has done most of the work already. As it states in their description, *"Impacket is focused on providing low-level programmatic access to the packets and for some protocols (e.g. SMB1-3 and MSRPC) the protocol implementation itself."*[10].
From usage of the Impacket library and the accompanying tools the author has had best result with the WMIExec tool for remote access, and therefore that is the one used in this project.

**Impacket WMIExec** WMIExec is a remote access tool from the Impacket suite that supports both clear-text and PTH authentication. Furthermore it has a semi-interactive shell accompanied, which can be used to upload and download files from the remote host. If we look deeper into the technical process, it works by using WMI to execute commands and SMB to upload/download files. Looking at the source code[11] we can get an overview of how it works.

1. A SMB connection is established to remote host using NTLM Authentication. The SMB connection is used every time a file needs to be uploaded or downloaded

2. If the command parameter is not set, it will do nothing and wait for input, otherwise it will use the WMI protocol to execute a command and get the result

3. In the case that an interactive shell is started, the tool will execute every command inputted by executing a remote command using the WMI protocol, and thereafter return the result when the remote command has executed

Now that we have a remote access to the host using either clear-text credentials or a hash, we can start the process of extracting memory form the LSASS process. As mentioned, this process will contain all credentials for currently active sessions.

**Dumping LSASS memory** Dumping memory of a process in Windows is a somewhat difficult tasks when you do not have a graphical interface. Using a graphical interface you can easily dump the memory of a process using Task Manager[13], but in the command line it is somewhat more difficult as no native ways exists. The solution is to use to tool Procdump from Microsoft's Sysinternal toolset.
With a WMIExec connection present we can do the following to dump and download the memory of a remote host

1. Start WMIExec with a remote connection to the remote host
2. Upload the Procdump tools to the remote host
3. Run Procdump with the parameters *-ma -accepteula lsass.exe debug.dmp* to save the LSASS process memory to debug.dmp
4. Download the debug.dmp file to our own machine
5. Delete Procdump and debug.dmp
6. Close the connection

#### 3.2.1 Local Security Authority Subsystem Service (LSASS) memory credential extraction

The LSASS process is a complex process with many usages as mentioned in section 3.1. The part we are interested in for this project is the credentials stored in memory. A lot of effort has gone into reverse engineering and studying the memory of this process, but the most interesting work has been done by Benjamin Delpy who created the very well known tool Mimikatz[3].

**Mimikatz** Mimikatz has a lot of features but is mostly known for being able to extract credentials from the memory Windows machines. Over the years many steps have been taken to make Mimikatz irrelevant, but it continues to be one of the most useful tools for Windows hacking. Mimikatz works on both a memory dump or live on the host, but in our case we are interested in working on a memory dump, as uploading Mimikatz to a host will in many cases be flagged by AV. For this project we are therefore mostly interested in the following commands from the sekurlsa[6] module[4]

**sekurlsa::minidump** The minidump command will switch Mimikatz into working on a memory dump instead of the current machine

**sekurlsa::logonpasswords** The logonpasswords will extract all credentials found in memory. This may include both clear-text and hashed passwords

One important note is that in order for Mimikatz to work properly on a memory dump, the architecture of the hosts must match, so that if the memory dump is from a x64 machine, the host doing the credential extraction must also be a x64 machine.

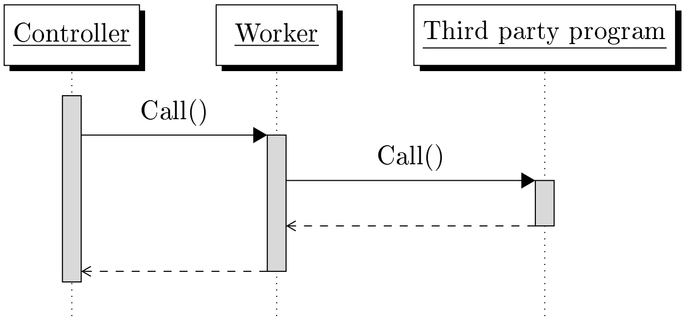


Figure 14: Test figure

# 4 Implementation

## 4.1 Technologies

### 4.1.1 ASP.NET Core

### 4.1.2 SignalR

### 4.1.3 VueJS

## 4.2 Considerations

### 4.2.1 Modularity

### 4.2.2 Ease of use

### 4.2.3 Traceability

## 4.3 Storage

# 5 Discussion

## 5.1 Ethics

# 6 Conclusion

[6] sekurlsa means Secure Local Security Authority (LSA)

## List of Figures

## References

[1] Aboba et al. *Link-Local Multicast Name Resolution (LLMNR)*. RFC 4795. IETF. URL: https://tools.ietf.org/html/rfc4795.

[2] BYT3BL33D3R. *Practical guide to NTLM Relaying in 2017 (A.K.A getting a foothold in under 5 minutes)*. URL: https://byt3bl33d3r.github.io/practical-guide-to-ntlm-relaying-in-2017-aka-getting-a-foothold-in-under-5-minutes.html (visited on 01/04/2018).

[3] Benjamin Delpy. *Mimikatz*. URL: https://github.com/gentilkiwi/mimikatz (visited on 01/07/2018).

[4] Benjamin Delpy. *Mimikatz*. URL: https://github.com/gentilkiwi/mimikatz/wiki/module-~-sekurlsa (visited on 01/08/2018).

[5] EvilMog. *Reversing MSCHAPv2 to NTLM*. URL: https://hashcat.net/forum/thread-5912.html (visited on 11/28/2018).

[6] Péter Gombos. *LM, NTLM, Net-NTLMv2, oh my! - A Pentester's Guide to Windows Hashes*. URL: https://medium.com/@petergombos/lm-ntlm-net-ntlmv2-oh-my-a9b235c58ed4 (visited on 11/28/2018).

[7] NetBIOS Working Group. *Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods*. RFC 1002. IETF. URL: https://tools.ietf.org/html/rfc1001.

[8] NetBIOS Working Group. *Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifiations.* RFC 1002. IETF. URL: https://tools.ietf.org/html/rfc1002.

[9] Hashcat. *Example hashes.* URL: https://hashcat.net/wiki/doku.php?id=example_hashes (visited on 01/04/2018).

[10] SecureAuth Labs. *Impacket.* URL: https://github.com/SecureAuthCorp/impacket (visited on 01/07/2018).

[11] SecureAuth Labs. *wmiexec.py.* URL: https://github.com/SecureAuthCorp/impacket/blob/master/examples/wmiexec.py (visited on 01/07/2018).

[12] Microsoft. *Account lockout threshold.* URL: https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/account-lockout-threshold (visited on 11/17/2018).

[13] Microsoft. *How to create a user-mode process dump file in Windows.* URL: https://support.microsoft.com/en-us/help/931673/how-to-create-a-user-mode-process-dump-file-in-windows (visited on 01/07/2018).

[14] Microsoft. *Microsoft NTLM - Message Syntax.* URL: https://msdn.microsoft.com/en-us/library/cc236639.aspx (visited on 12/24/2018).

[15] Microsoft. *Microsoft TCP/IP Host Name Resolution Order.* URL: https://support.microsoft.com/en-us/help/172218/microsoft-tcp-ip-host-name-resolution-order (visited on 11/20/2018).

[16] Microsoft. *Mitigating Pass-the-Hash (PtH) Attacks and Other Credential Theft Techniques.* URL: https://download.microsoft.com/download/7/7/a/77abc5bd-8320-41af-863c-6ecfb10cb4b9/mitigating%20pass-the-hash%20(pth)%20attacks%20and%20other%20credential%20theft%20techniques_english.pdf (visited on 12/12/2018).

[17] Microsoft. *NTLM Over Server Message Block (SMB).* URL: https://msdn.microsoft.com/en-us/library/cc669093.aspx (visited on 11/15/2018).

[18] Microsoft. *Securing Privileged Access Reference Material.* URL: https://docs.microsoft.com/en-us/windows-server/identity/securing-privileged-access/securing-privileged-access-reference-material (visited on 01/04/2018).

[19] Microsoft. *SSPI Model.* URL: https://docs.microsoft.com/en-us/windows/desktop/secauthn/sspi-model (visited on 12/12/2018).

[20] BrandonWilson - Microsoft. *The Importance of KB2871997 and KB2928120 for Credential Protection.* URL: https://blogs.technet.microsoft.com/askpfeplat/2016/04/18/the-importance-of-kb2871997-and-kb2928120-for-credential-protection/ (visited on 01/04/2018).

[21] The Cable Guy - Microsoft. *Link-Local Multicast Name Resolution.* URL: https://docs.microsoft.com/en-us/previous-versions//bb878128(v=technet.10) (visited on 11/20/2018).

[22] Mockapetris. *Domain Names - Implementation and specifiation.* RFC 1035. IETF. URL: https://tools.ietf.org/html/rfc1035.

[23] Ronald Tschalär. *NTLM Authentication Scheme for HTTP.* URL: https://www.innovation.ch/personal/ronald/ntlm.html (visited on 01/03/2019).

[24] Georgia Weidman. *Scenario-based pen-testing: From zero to domain admin with no missing patches required.* URL: https://www.computerworld.com/article/2843632/security0/scenario-based-pen-testing-from-zero-to-domain-admin-with-no-missing-patches-required.html (visited on 11/15/2018).

[25] Vladimir Gladkov - DZone - Integration Zone. *Practical Fun with NetBIOS Name Service and Computer Browser Service.* URL: https://dzone.com/articles/practical-fun-with-netbios-name-service-and-comput (visited on 11/28/2018).