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. 23

AV Antivirus. 6, 25

DC Domain Controller. 7, 24

DI Dependency Injection. 33, 37

GPU Graphics Processing Unit. 16

HTML Hypertext Markup Language. 34

HTTP HyperText Transfer Protocol. 15, 16, 20–22, 26, 29, 31, 33, 38

I/O Input/output. 32

IDD Interface driven development. 29, 32, 33, 37

IDS Intrusion Detection System. 6

JSON JavaScript Object Notation. 3, 27, 31, 32

KDC Key Distribution Center. 24

LLMNR Link-local Multicast Name resolution. 6–9, 12–15, 22, 26, 33, 38

LSA Local Security Authority. 25

LSASS Local Security Authority Subsystem Service. 22–26, 29–31

NBNS NetBIOS Name Resolution. 6–10, 12, 15, 22, 26, 33, 38

PTH Pass-the-hash. 23, 24

RDP Remote Desktop Protocol. 22, 23

SIEM Security Information and Event Management. 6

SMB Server Message Block. 15, 16, 18–20, 22, 24, 26, 29, 31, 33, 38

SSO Single Sign-On. 15, 22

SSP Security Support Provider. 15, 18, 20

SSPI Security Support Provider Interface. 15

WINRS Windows Remote Management. 23

WINS Windows Internet Name Service. 9

WMI Windows Management Instrumentation. 23, 24

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 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)[36] to gain crackable hashes which can be bruteforced offline. Hereafter a number of different techniques and exploits can be used for lateral movement, 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.

During a pentest it is usually required 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 problem mentioned in section 1.1 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 a 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?[15]	-	-	+	+	+
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 are actively used in real life penetration tests. Table 1 scores each method according to their pros and cons, and here the reader can clearly see 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[29] and not monitored

[1] This method can be very time consuming

correctly. It is important to mention that there exists a situation and place for every method, but the chosen method of spoofing is what will suit this project the best.
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.[22] 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 the packets sent are well-formed and conform to the standards of the protocol, the *Client* will register *Server1* to have the IP address of *Attacker*, and thereby sending all packets intended for *Server1* to *Attacker*.

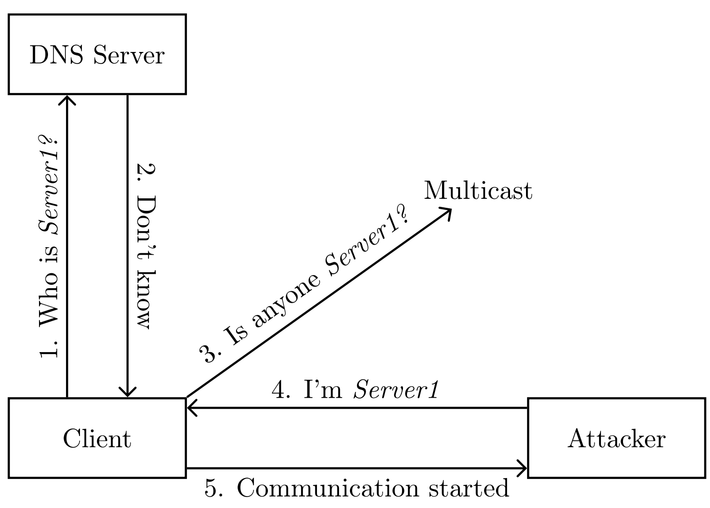
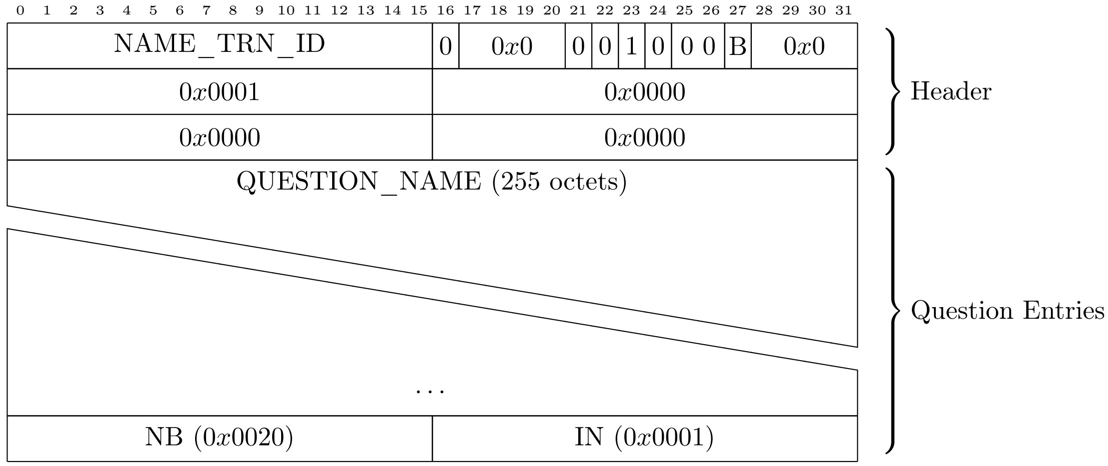


Figure 1: How an attacker spoofs the Name Resolution protocols in order to receive traffic intended for other hosts

In Windows, two different Name Resolution protocols are used and active by default on all modern Windows versions. LLMNR and NBNS work side by side, unless specifically turned off, which is not the case by default. In section 2.1.2 and 2.1.2 the protocols are analyzed in detail and the attacks are described.
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[9], contains much more than Name Resolution, but for spoofing purposes we only need to look at Name Resolution. RFC 1002[10] contains detailed technical specification as to how Name Resolution is implemented in NetBIOS.
NBNS has many other features which are unrelated for this project, 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

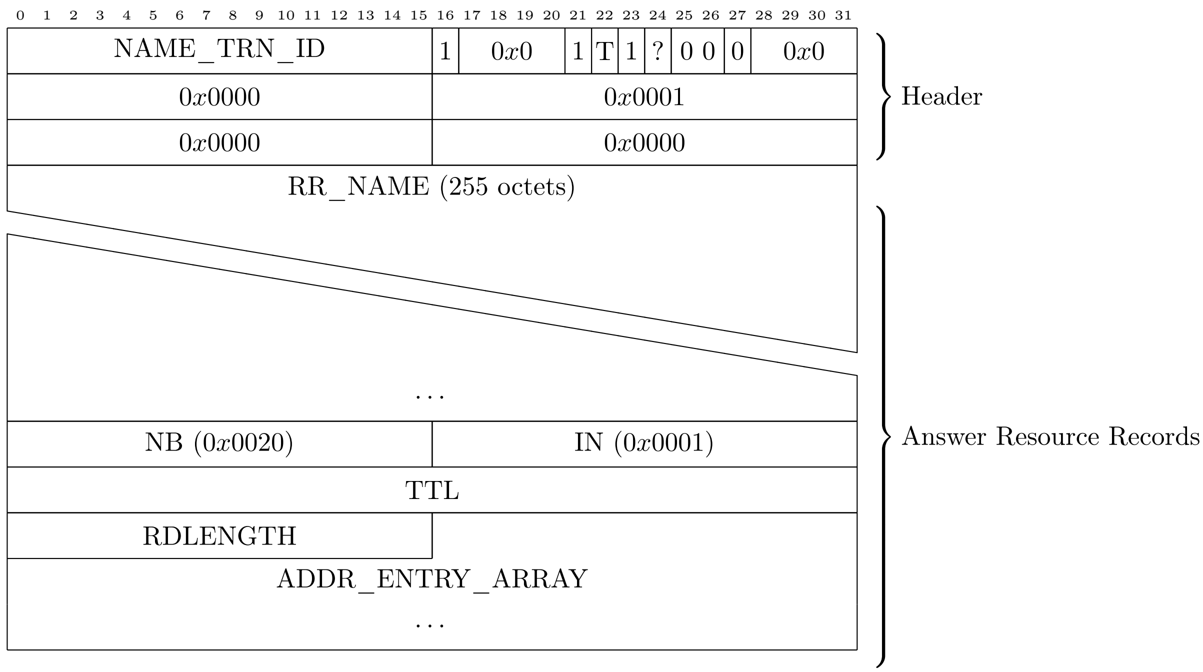
NAME_TRN_ID is the transaction ID for the Name Service Transaction

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

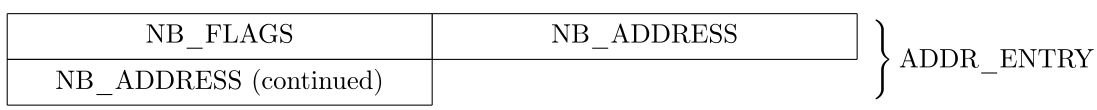
Figure 2: NetBIOS Name Resolution (NBNS) Name Query Request[10, sec. 4.2.12]

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'[40], 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



Where each ADDR_ENTRY has the following format:



And

T is whether or not the data is truncated

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(The IP of the attacker)

Figure 3: NetBIOS Name Resolution (NBNS) Name Query Response[10, 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 FDEFFCFGEFFCDBCACACACACACACACACACACA respectively. According to the information gained from figure 3, 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. This is what Windows uses by default), **RDLENGTH** to 6 ($0x0006$. 2 bytes for **NB_FLAGES** and 4 bytes for **NB_ADDRESS**) 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 figure 4,which will result in a successful spoof of the NBNS protocol, such that traffic from *client* intended for *server1* is sent to *attacker*.

```
0000   d0 56 01 10 00 01 00 00 00 00 00 00 20 46 44 45   ĐV.......... 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..ĐV....
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 then 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-local 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[30].

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.

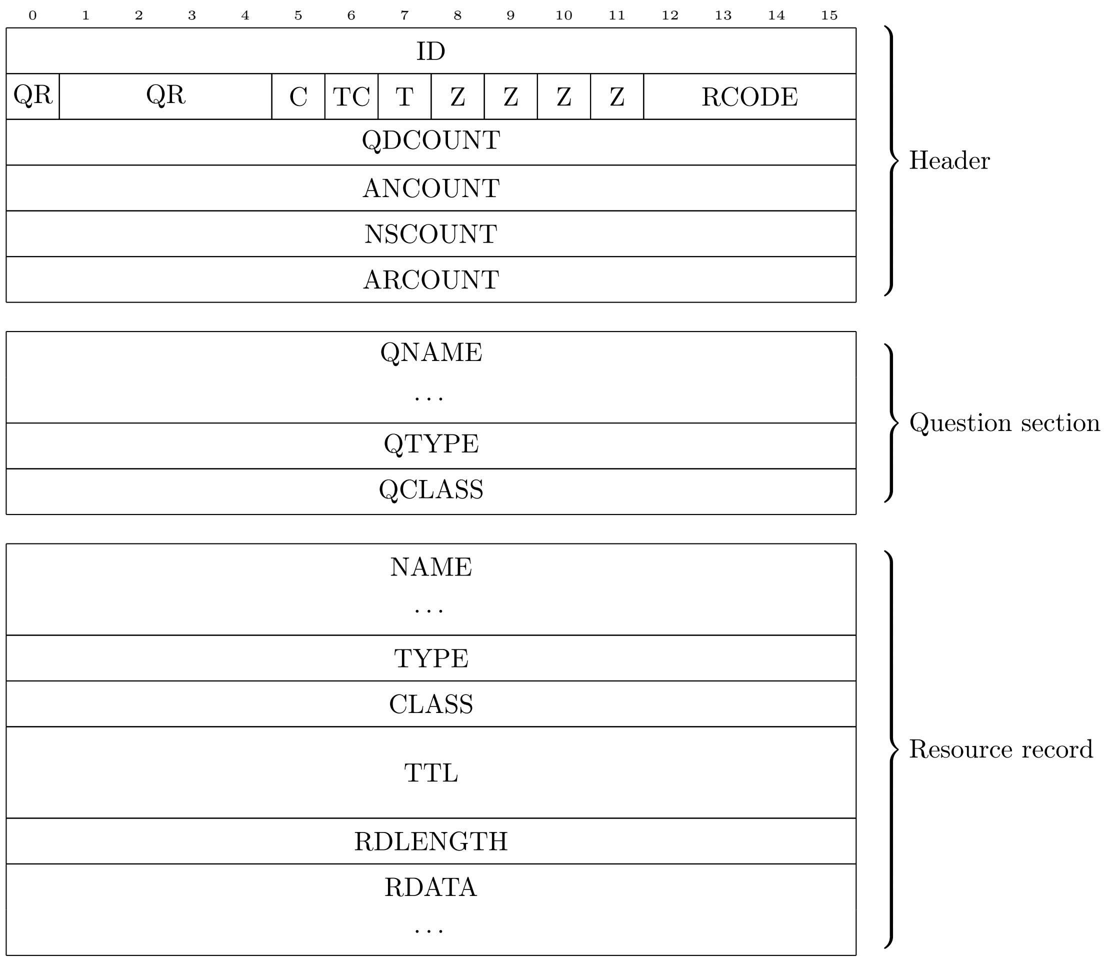


Figure 6: Link-local Multicast Name resolution (LLMNR) packet specification[1][30]

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][30]

This report will not explain the packet details in full[3], but will focus on the parts necessary to spoof a LLMNR response.
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

[3]The header is not explained as all fields contain static values as it can be seen in the specification[1]

Figure 7 and 8 shows a valid request and the corresponding spoofed response for a name resolution for *server1*. After responding to the request with the proper response we would have successfully spoofed the LLMNR protocol and redirected traffic intended for server1 to our host(The attacker).

2.2 Credential acquiring

After successfully spoofing either a NBNS or LLMNR request we have now succeeded in imposing 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. The security models works as a Single Sign-On (SSO) solution that allows users to easily authenticate to various services using their cached credentials[27]. We can use this to our advantage by implementing a Security Support Provider (SSP) in fake SMB and HTTP services. There exists a couple of different SSP's including Negotiate, NTLM, Kerberos, Digest SSP and others[26]. 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[8]

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 converted to a DES key and 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[23] 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[7].

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

Besides being a hash, NTLM is also the name of 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[21]. 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 can be done. 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* with a null byte (0x00) in the end[35]. 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 service 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 is 8 arbitrary bytes chosen by the server

FLAGS is 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	$x01$	$x82$	0	0
DOMAIN ...							
USER ...							
HOST ...							
LM ...							
NT ...							

Where

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

CHALLENGE is 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 from this message. The full flow of this process can be seen on figure 11 which starts after a host has successfully been spoofed to redirect traffic to our host.

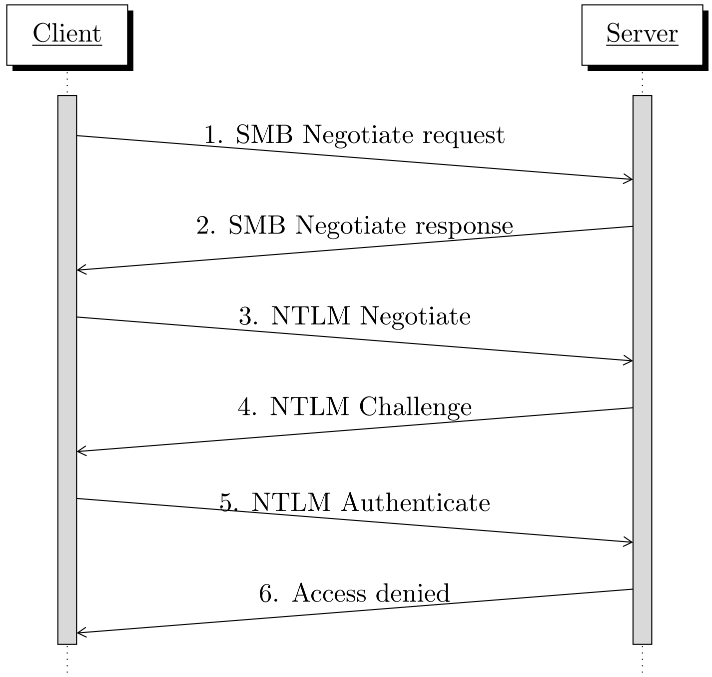


Figure 11: SMB NTLM authentication[24]

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 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[4], or any other authentication for that matter.

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

[4]Kerberos authentication could also be used, as we would get a ticket which can also be bruteforced. The hash algorithm is stronger though, so NTLM is preferred

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, done by encapsulating the NTLM authentication messages into the HTTP protocol. 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[5], 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. This hash can be cracked offline using various techniques.

[5] 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

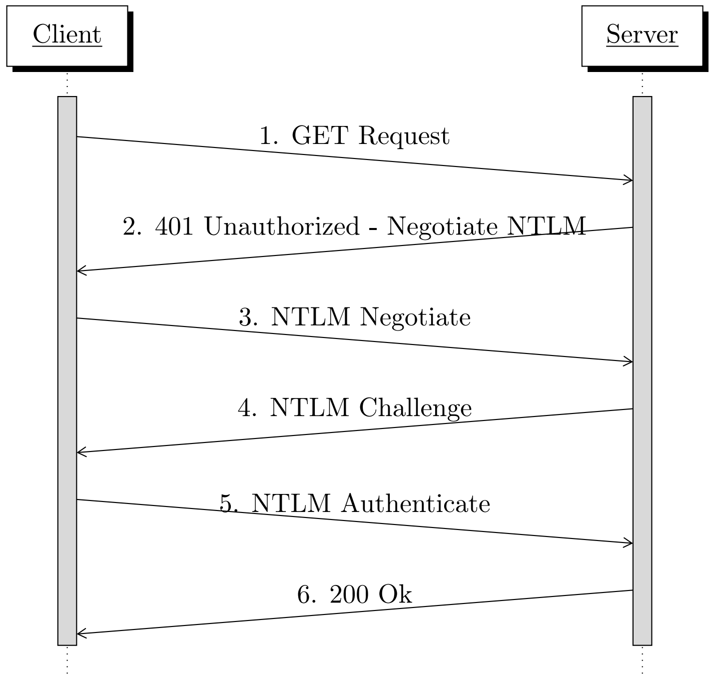


Figure 13: HTTP NTLM authentication[35]

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.

The method has been tested on all major browser (Internet Explorer, Chrome, Firefox and Edge), and works flawlessly in all of them.

3 Attack methods

After successfully spoofing a victim by abusing LLMNR and NBNS and getting a NetNTLMv2 hash by running fake SMB and HTTP services, 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[12] 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[4] where you construct your HTTP or SMB services to relay the credentials to another server/host.[6]
After having 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's 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 uncertain way of achieving Domain Admin privileges, as most known vulnerabilities are fixed quite easily and we are not in the position of possessing zero day vulnerabilities for Windows. Luckily there are many other methods to achieve lateral movement in a Windows domain, which this project will explore further in the following sections. Analyzing and discussing all the different types of attacks is not the focus of this 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 an user 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[7]. The feature of storing password in the memory is actually what SSO uses to sign-in

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

[7]LSASS does not always save the credentials in memory, it depends on the type of authentication and how it happens[25]

automatically to services, and it is a very essential part of why spoofing 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 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 the 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[28]. 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 an interactive session on another host, where the session 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)**[33]. 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 versions of Windows, 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 18 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)[8].
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 official 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."*[13].
From common usage of the Impacket library and the accompanying tools in the Impacket suite, 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 to/from the remote host. If we look deeper into the technical aspects, it works by using WMI to execute commands and SMB to upload/download files. Looking at the source code[14] we can get an overview of how it works.

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

2. If the command parameter is not set, the tool 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 from the LSASS process. As mentioned in section 3.1, 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[16], but having only access to the command line it is somewhat more

[8]In Windows this is usually a DC

difficult as no native ways exists. One solution is to use the tool Procdump from Microsoft's Sysinternal toolset.
With a WMIExec connection present we can do the following to dump and download the memory of the LSASS process 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[5].

Mimikatz Mimikatz has a lot of features but is mostly known for being able to extract credentials from the memory of 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 can work on both memory dumps or directly on the host, and in this project 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[9] module[6]

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.

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

4 Implementation

In the previous sections the different protocols and techniques were described in depth. The next logical step is, of course, to implement this into a working piece of software that can be used in real life situations. To do this we need to establish some requirements for the software. As the title states, the software needs to be easy to use and contain a high level of traceability, such that timeline of actions can be created easily. Other considerations such as security should also be taken into account, for example passwords should not be saved in clear text on the disk, as we are not always sure that the host is properly secured. When deploying the software on foreign networks, we do not want to alter the state of security that the network is currently in, as we can not be sure that the network does not contain malicious actors.

4.1 Requirements

This project is mostly designed for the authors own needs and workflow, so the requirements are not strictly set or defined. This only works because it is a one-man project with only one developer. In case multiple developers were to work on the project simultaneously both requirements and other technical parts should be more strict and not leave it up to the developer to decide. The most general requirements are mentioned below, split into functional and technical requirements, as this will give a satisfactory overview of the project and its technological choices.

Functional requirements

1. Should be designed with ease of use in mind
2. Must save actions to the disk in an easy to read format
3. Must implement both a LLMNR and NBNS spoofer
4. Must implement both a SMB and HTTP service for credential acquiring
5. Must be able to dump and download the LSASS process memory from another host on the network
6. Must be able to extract NT hashes and clear-text from said memory dump
7. Must be able to show a list of credentials in a graphical interface
8. Must be able to show a list of hosts in a graphical interface
9. Must be able to show a log of all actions done
10. Must update interface seamlessly, ie the user should not manually update the interface to see changes
11. Sensitive information must be encrypted when saved on the disk

Technical requirements

1. The software must be developed in C# using ASP.NET Core
2. The software should strive to be cross platform
3. The graphical interface must be web-based
4. WebSockets should be used to create a real-time app
5. The code must be modular
6. The code should be written with high cohesion and low coupling, such that the graphical interface can easily be exchanged
7. The code must be multi-threaded so that multiple actions can be done simultaneously
8. Data must be saved in JSON files on the disk
9. A C# annotation should be used to specify fields to encrypt when saving to disk
10. Saving data to disk should not disrupt any other actions
11. The frontend should use a JavaScript framework such as React, Angular or Vue

To concretize the requirement list the software should be a web-app developed in C# using ASP.NET Core. The Web app should be fluid and work seamlessly, and this should be achieved by using WebSockets combined with a JavaScript framework to ensure that the graphical interface is updated immediately without any user actions. The graphical interface in the form of a web-app must be easily exchangeable such that it can be changed without extensive refactoring of the underlying backend codebase. Additionally the underlying backend (which from hereon is mentioned mentioned as the *worker*) should implement the spoofing protocols weaponized in section 2 (Initial foothold) and the attack methods analyzed in section 3 (Attack methods).

4.2 Technologies

Before the actual implementation can be programmed and explained we need to discuss the different technologies used in the implementation and why they were chosen. This is briefly done in the next sections to provide the reader with the arguments and decisions behind technology choices.

4.2.1 ASP.NET Core

Almost all programming languages have their own web framework and they mostly work the same. There are definitely pros and cons to all frameworks, but the author has a large background in C# and the .NET framework, and therefore ASP.NET Core was the obvious choice. The advantage of ASP.net Core is that it is cross platform[19] and the fact that it uses C#, which in the authors opinion, is a very good option for a modern and safe programming language.
Furthermore, when pentesting Windows networks it makes sense to use a programming language and framework developed by the same company.

4.2.2 SignalR

As mentioned in the requirements sections (4.1), the web-app needs to use WebSockets for two-way communication between the client and server to ensure that the user does not need to refresh the graphical interface manually. WebSockets is a relatively new technology first seen in the Chrome webbrowser in version 43 which was released in 2015[31]. For this projects WebSockets should be used to update the graphical interface in real time with new informations such as log messages, new/changed users and hosts. C# supports WebSockets natively using the `Microsoft.AspNetCore.WebSockets` package but external libraries exists to aid developers in using WebSockets. One such library is SignalR[20] which is an open-source library that simplifies the creation of real-time web-apps by using WebSockets, Server-Sent events and long polling. SignalR was chosen for its simplicity, popularity and its ecosystem with a lot of active users.

4.2.3 VueJS

When developing a modern and interactive web UI using a JavaScript framework is almost required. A proper framework will make the development process easier and in return create a more stable and fluid frontend. When developing an live web app a frontend will also make it easier to ensure that the client and server side is synchronized properly. So that there is no difference in the data possessed by the server and the data shown in the graphical interface.
There exists a wide range of JavaScript libraries with each one having their strengths and weaknesses. The two most popular ones are Angular and React closely followed by Vue[11]. React and Angular is primarily developed by Facebook and Google respectively, and have a large community surrounding them. Vue on the other hand is the small underdog created by Evan You as an open-source project with the goal of creating a lightweight JavaScript library that could compete with the likes of Angular and React. Vue is often described as the best parts of Angular and React but without all the bloat that comes with such big frameworks. That, and the fact that the author has previously worked with Vue, is the choice for using Vue for this project. Vue works very well for small web-apps with a small developer team, where the code structure is somewhat simple.

4.3 Considerations

When developing software of this complexity a solid foundation is important to be able to continue development on it. Even though the most optimal techniques and attack vectors have been chosen, it is very plausible that other non-considered techniques should be implemented later on. This requires the foundation to be able to support further development on the project, and the project should therefore strive to be extendable. This is also the reason for the separation of the worker process and the graphical interface in the form of the web app. Separating these two will make it easier to develop both simultaneously as long as they share the same interface.

4.3.1 Interface driven development (IDD)

The whole project is developed with the Interface driven development (IDD) technique[2][3] in mind. When using interfaces properly in any object oriented language you, at least when you do it correct, you gain a few advantages such as

1. Easy and powerful unit-testing
2. Better architecture, as one has to design the interfaces and correlations beforehand which requires more analysis
3. Maintainable code. All code can essentially be changed or replaced as long as it still implements the same interface

4.3.2 Modularity

If we look back at section 2 and 3 we can clearly see how a modular approach can be useful in the development of the worker. For example spoofing, services (SMB and HTTP) and LSASS dumping is one module each. With a common interface `IModule` that each module needs to implement, we can streamline the modules easily and allow for new ones to be added without much hassle. There also needs to be an easy way to register new modules in the worker, so that extending the functionality is easy.

Common interface There needs to be a common interface for each module, denoted `IModule`. This module should only have one field Name, as seen on listing 1, such that a module can be easily identified by its name. This is useful when logging actions made by the different modules.

```
public interface IModule
{
    string Name { get; }
}
```

Listing 1: Interface IModule

Type of modules The analysis done shows us that we both need persistent modules such as spoofing and services, and “action” modules such as dumping LSASS. From this we can create two interfaces. **IPersistentModule** that is supposed to run in the background and **IActionModule** that is run when the user chooses to run it. The two interfaces can be seen on listing 2 and 3. All module types needs to implement the IModule interface to make sure implementations hereof conforms to the module interface.

```
public interface IPersistentModule : IModule
{
    Task Run();
    Task Stop();
    bool IsEnabled { get; }
}
```

Listing 2: Interface IPersistentModule

```
public interface IActionModule : IModule
{
    Task Run(Target target, User user);
}
```

Listing 3: Interface IActionModule

Adding new modules Adding new modules should be easy and straight forward. As a solution C# has features to load external assemblies into the running program[17], but this exposes our application to certain security risks. For example a malicious user could load their own assembly instead of our module, and thereby achieving code execution within our “domain”. This can be avoided with certifications and validation hereof, but require much stricter control and additional development. Seeing as this is a project where modules are mostly static, ie. they are not added ad hoc, it is not a good solution to the problem.
Therefore a not quite optimal solution was chosen where modules are added by hand in the worker in the method *RegisterModules*. So the worker will have

the responsibility to register modules and start/stop them. It also has the responsibility to synchronize they action module with the graphical interface through the *IWorkerController*. It is this interface that the graphical interface needs to implement and the interface that permits the graphical interface and the worker to communicate.

4.3.3 Traceability

When executing pentests of Windows domains certain documentation is a big part of the assignment. Actions needs to be logged, and after completion a list of servers compromised in order to gain Domain Admin privileges has to be provided in the correct order with timestamps. Even though every action is usually documented thoroughly certain actions can slip and be hard to remember afterward. So therefore the project should strive to document and trace every action made. So whenever a credential is acquired, it should be logged from where it came (HTTP, SMB or LSASS dump), when it happened and, if applicable, which host it came from.
To accommodate this challenge the program needs to save every action made in a persistent way on the disk. As mentioned in section 4.1 (Requirements) this also needs to support encryption of individual fields in order to restrict unauthorized access to confidential data. As the requirements also state, the data must be saved in the form of JSON files on the disk directly, such that third party databases are not needed.

4.4 Storage

As mentioned in section 4.1 (Requirements) the data should be saved on disk in the JSON format and have the capability to encrypt individual fields. So this part of the project consists of two different aspects

1. A JSON serializer
2. A way to save the JSON object to disk

The .NET standard for JSON serialization is Json.NET from Newtonsoft[32], which will also be used by this project. Newtonsoft has the necessary features to implement encryption of individual fields[10]

4.4.1 Efficient persistent JSON data storage

When saving data to JSON in C#, you basically convert a C# object into its corresponding JSON structure, which is called serializing. The problem with this is, that you cannot serialize parts of an object. For example if one serialize

[10] You should never roll your own crypto. Therefore most of this code is taken directly from `https://stackoverflow.com/questions/29196809/how-can-i-encrypt-selected-properties-when-serializing-my-objects`, which uses standard encryption algorithms and has been verified by multiple people

a list of strings, and then decide to change on of the strings one cannot simply change part of the JSON object. In that case you would have to serialize the whole list to a JSON structure once again. This can lead to certain bottlenecks when dealing with data that updates often. The implementation seeks to overcome this hurdle by having two layers of storage consisting of both a volatile in-memory storage and a persistent disk storage.
When dealing with a highly threaded program with large amounts of Input/output (I/O), concurrency is very important. When the process of serializing an object before saving it to disk is also a time consuming process, certain aspects of the saving logic needs to be optimized. For this project a custom JSON storage provider was implemented with the following things in mind.

- The storage provider is implemented using IDD, to allow for other storage methods than JSON like databases, in-memory databases or similar.
- Each object is mapped to one datastore which is mapped to one file. For example the object User is mapped to the file User.json
- All I/O is firstly done directly in memory so latency is kept at minimum. Every memory action is then synched to the corresponding data file on disk in a separate "commit" thread.
- Semaphore locks are implemented such that only one thread can access the underlying file at a time
- The file itself is never locked, unless written to, to allow manual editing and concurrent reads.
- Singleton pattern is implemented to make sure that there is never registered more than one datastore for each object.
  - The singleton pattern is implemented using the `System.Collections.Concurrent.ConcurrentDictionary` to ensure it is thread-safe
- A `System.ComponentModel.FileSystemWatcher` is started for every datastore to synchronize manual changes done to the files

If we look at the class diagrams from appendix A.2 (Web class diagram) and A.3 (Data objects class diagram) we can see how this is all combined with the implementation of `JsonDataStore` and `JsonDataStoreObject<T>`.

JsonDataStore implements the `IDataStore` interface and handles the registration of stores and allows the code to get already registered stores.

JsonDataStoreObject<T> implements the `IDataStoreObject` interface with the generic parameter T which is a `IDataObject`. All data objects implement the interface `IDataObject`. This class also contains all the logic for saving to files/fetching from files and handle the in-memory mapping of the file.

User is a data object containing username, password, domain, timestamp, passwordtype and hash, where password and hash is encrypted when saved on disk.

Target is a data object containing hostname, ip, dumped boolean, dumpedTimestamp and added timestamp.

LogEntry is a data object containing timestamp, name, message and parameters. It also contains a formatted message and a timestamp string for easier formatting when shown in the graphical interface.

4.5 Code Structure

With the considerations from section 4.3 in mind and adhering to the requirements from section 4.1 the structure seen on appendix A.1 (Worker class diagram) and A.2 (Web class diagram) is the final structure of the application. Overall the application exists of two distinctive parts, the `Worker` and the implementation of `IWorkerController`, `WebController`. When a `Worker` is initialized a `IWorkerController` is passed on as a parameter in the constructor. The project uses the built in version of Dependency Injection (DI)[34] for ASP.Net Core to achieve inversion of control, such that the `Worker` can control its `IWorkerController`. DI also adheres to the principles of IDD.

4.5.1 `Worker`

The primary task of the `Worker` is to register and handle modules in their lifetime. Two types of module exists right now, but the implementation is structured such that new ones can be added easily.

`IPersistentModule` is a module that is meant to run all the time in its own thread. The module lifetime is handled by the `Worker`, and therefore requires to implement both `Start` and `Stop` methods in order to be controlled.

Persistent modules are then currently split in two in the form of **spoofers** and **Servers (or services)**. As the logic for most **spoofers** are the same, they are all controlled by `SpooferCore` and inherits from the base class `BaserSpoofer` to keep redundant code at a minimum. The logic for **services** vary much though, and therefore they do not inherit from a base class.

`LLMNRSpoofer` **and** `NBNSSpoofer` is the implementation of the LLMNR and NBNS spoofers as analyzed in section 2.1 (Spoofing)

`HTTPServer` **and** `SMBServer` is the implementation of the HTTP and SMB servers/services as analyzed in section 2.2 (Credential acquiring)

IActionModule is a module that is meant to run on demand, which means that it only implements a `Run` method from the `IActionModule` interface. An example of such a module is the `LsassDumpTool` which is the attack explained in section 3.2

4.5.2 WebController

`WebController` is the implementation of `IWebController`. The `Controller`'s job is to handle the actions that the worker generates. For example this is logging or new credentials acquired. It is up to the `Controller` to make sure that data is saved and displayed properly to the user. This is done to separate the raw business logic from the graphical logic.

Hubs is, as stated by the SignalR documentation, a high-level pipeline that allows a client and server to call methods on each other[20]. Currently three hubs exists, `UserHub`, `TargetHub` and `LogEntryHub`, each corresponding to one type of data. The hubs are connected to by the Vue frontend and handles interaction between the user and the WebController.

HubActions is a middleware between the `WebController` and the `Hubs` and they all implement the interface IHubActions<T> where `T` is a `IDataObject`. They are implemented to get a single connection to hubs instead of having the WebController handling Hub connection. Currently three HubActions are implemented, `UserHubActions`, `TargetHubActions` and `LogEntryHubActions`

WorkerSettings is the implementation of `IWorkerSettings` to handle all settings for the `Worker`. The implementation uses the `Microsoft.Extensions.Configurations` library to load configurations from the `appsettings.json` file.

4.5.3 Vue JavaScript frontend

The frontend is built with the Vue JavaScript framework[38] and uses the state management library Vuex[39] to act as a centralized store for the three data objects (Users, targets and log entries).
Vue includes the ability to declare reusable components and this feature was used for each of the data objects, so that each object has its own component that includes both logic and the Hypertext Markup Language (HTML) view itself. Therefore the frontend consists of three components, a Vuex store and a app component to bind it all together.

Vuex store The store contains a dataset for each of the three data objects. Furthermore each object also has a `isLoading` boolean to show a loading page when data is loading. For each action done to a dataset a mutation method is implemented. An example of an implemented mutation can be seen on listing 4

```
[EDIT_USER](state,user) {
    var element = state.users.data.find(function (element) {
        return element.id = user.id;
    });
    element.username = user.username;
    element.hash = user.hash;
    element.isClearText = user.isClearText;
    element.hashcatFormat = user.hashcatFormat;
},
```

Listing 4: `EDIT_USER` mutation from the Vuex store

Data object components are all structured in the same way. SignalR requires each hub to have its own connection, so when a component is initialized a connection to the designated hub is started as well. The component then maps the necessary data from the Vuex store to itself. For example the user component will map the Vuex store's user data object and so forth. This ensures that all components always modify and read from the same dataset.

Additionally, all components have listeners for the currently implemented hub actions such as get and edit, that allows the server and client to synchronize automatically when changes occur.

4.6 Execution flow

To get a better overview of how the software itself is structured one can look at the sequence diagram on figure 14 showing the implementation `Microsoft.Extensions.Hosting.IHostedService.StartAsync`. The `Microsoft.Extensions.Hosting.IHostedService` interface is the default way of running background tasks in .NET Core[18]. In this project the spoofers and the services needs to run as a background task as they need to listen for network requests separately from the rest.

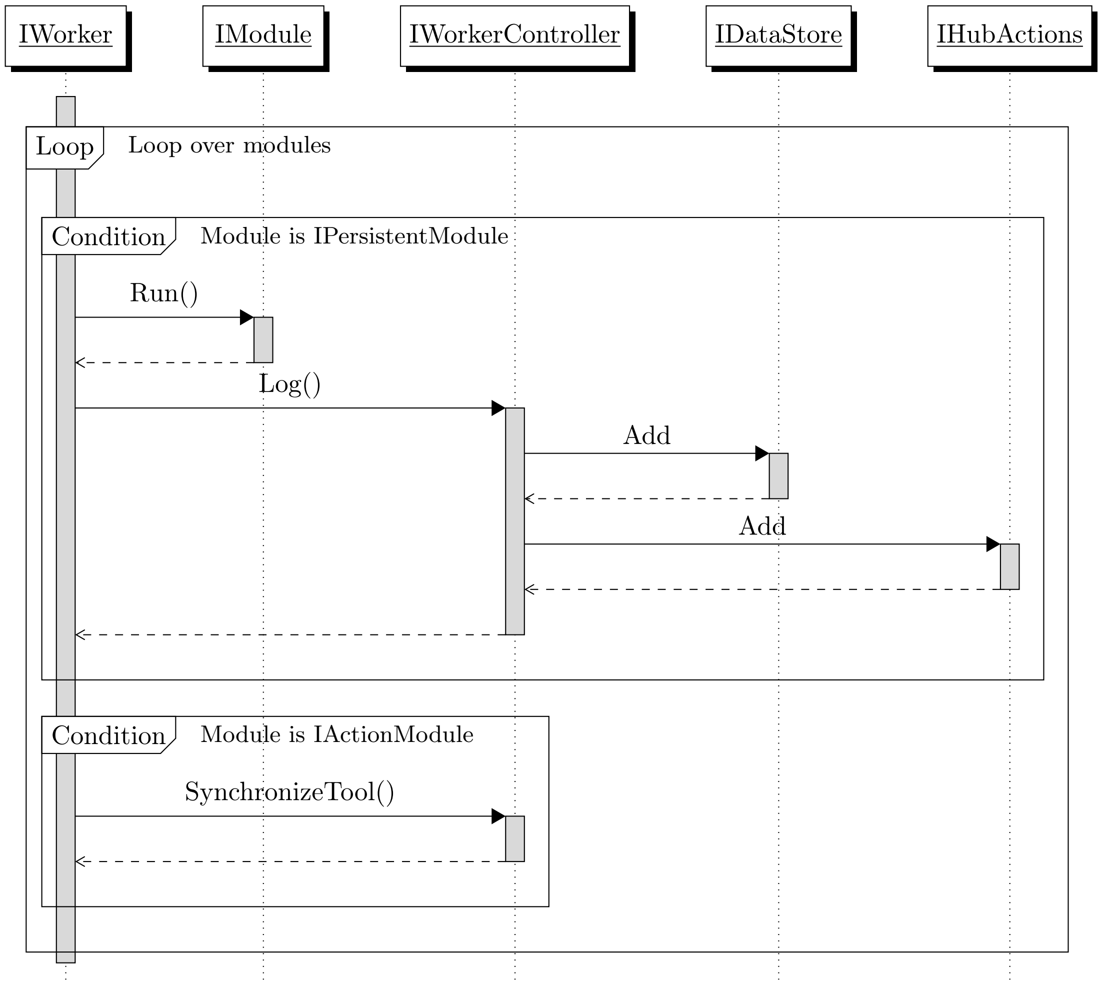


Figure 14: Sequence diagram of the `Microsoft.Extensions.Hosting.IHostedService.StartAsync` method implemented by `Worker`

When starting the background tasks the logic is split into two parts, one for persistent modules and one for action modules. On persistent modules a thread is spun up and the `Run()` method is executed in that thread. This gives the worker, not the module, control over the thread and can stop it when needed. When persistent modules are started a log message is sent from the `Worker` to the registered `IWorkerController` which then first hands it to the datastore for persistent storage and thereafter adds it to the designated hub. Currently three hub exists, UserHub, LogEntryHub and TargetHub.

Action modules are special in the way that they need to be run on demand and not as a background task. Therefore the `Worker` needs a way to tell the

`IWorkerController` which actions exists. This is done using the `SynchronizeTool()` method which should be implemented in the implementation of `IWorkerController`.

4.7 Testing

The project has been implemented with a focus on IDD which makes it much easier to unit test your software by creating mocks of interfaces. Furthermore the DI pattern ensures that the mocks can be injected into the classes being tested. In this project the unit test revolves around the `JsonDataStore` as this is one of the most critical places of the project. One such test can be seen on listing 5 where the integrity of the data is tested and whether or not the data was saved to the correct file.

```
[Fact]
public void TestJsonIntegrity()
{
    var targetStore = DataStore.Get<Target>();
    targetStore.Add(Target);
    var target = targetStore.Get(Target.Key);
    var savedJson = File.ReadAllText(string.Format("{0}/{1}", Path,
        "Target.json"));
    var generatedJson = JsonConvert.SerializeObject(new List<Target> { target },
        Formatting.None);
    Assert.Equal(savedJson, generatedJson);
}
```

Listing 5: Test method of the JsonDataStore to test the integrity of the data files

Most of the other testing done for this project consisted of manual testing of the required functionality such as the spoofers and the services. When implementing specifications it can be hard to do unit testing initially, as you need to act on the way the operating system reacts, and not so much on the exact specification. Wireshark[37] was mostly used for this, as it can parse the packets correctly and show when the packets does not adhere to the specification.

5 Discussion

5.1 Ethics

6 Conclusion

List of Figures

List of source codes

References

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

[2] Arash. *Interface Driven Development (part 1)*. URL: `https://medium.com/@aaraashkhan/interface-driven-development-part-1-b7c7011860c0` (visited on 01/11/2018).

[3] Arash. *Interface Driven Development (part 2)*. URL: `https://medium.com/@aaraashkhan/interface-driven-development-part-2-4406ee738514` (visited on 01/11/2018).

[4] 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).

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

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

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

[8] 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).

[9] 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.

[10] 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.

[11] Vijay Singh - Community Manager @ Hackr.io. *Top Javascript Framework used in 2018*. URL: https://www.codementor.io/vijaysingh782/top-javascript-framework-used-in-2018-oem84fncp (visited on 01/10/2018).

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

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

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

[15] 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).

[16] 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).

[17] Microsoft. *How to: Load Assemblies into an Application Domain*. URL: https://docs.microsoft.com/en-us/dotnet/framework/app-domains/how-to-load-assemblies-into-an-application-domain (visited on 01/11/2018).

[18] Microsoft. *Implement background tasks in microservices with IHostedService and the BackgroundService class.* URL: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/background-tasks-with-ihostedservice (visited on 01/15/2018).

[19] Microsoft. *Introduction to ASP.NET Core.* URL: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2 (visited on 01/09/2018).

[20] Microsoft. *Introduction to ASP.NET Core SignalR.* URL: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-2.2 (visited on 01/10/2018).

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

[22] 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).

[23] 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).

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

[25] 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).

[26] Microsoft. *SSP Packages Provided by Microsoft.* URL: https://docs.microsoft.com/en-us/windows/desktop/secauthn/ssp-packages-provided-by-microsoft (visited on 01/12/2018).

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

[28] 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).

[29] 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).

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

[31] Mozilla. *The WebSocket API (WebSockets)*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (visited on 01/09/2018).

[32] Newtonsoft. *Json.NET - Newtonsoft*. URL: https://www.newtonsoft.com/json (visited on 01/11/2018).

[33] Stackoverflow. *How to execute a Windows command on a remote PC?)* URL: https://stackoverflow.com/questions/11935695/how-to-execute-a-windows-command-on-a-remote-pc# (visited on 01/12/2018).

[34] Scott Addie Steve Smith and Luke Latham - Microsoft. *Dependency injection in ASP.NET Core*. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2 (visited on 01/15/2018).

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

[36] 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).

[37] Wireshark. *Wireshark - Go deep*. URL: https://www.wireshark.org/ (visited on 01/15/2018).

[38] Evan You. *Vue.js*. URL: https://vuejs.org/ (visited on 01/16/2018).

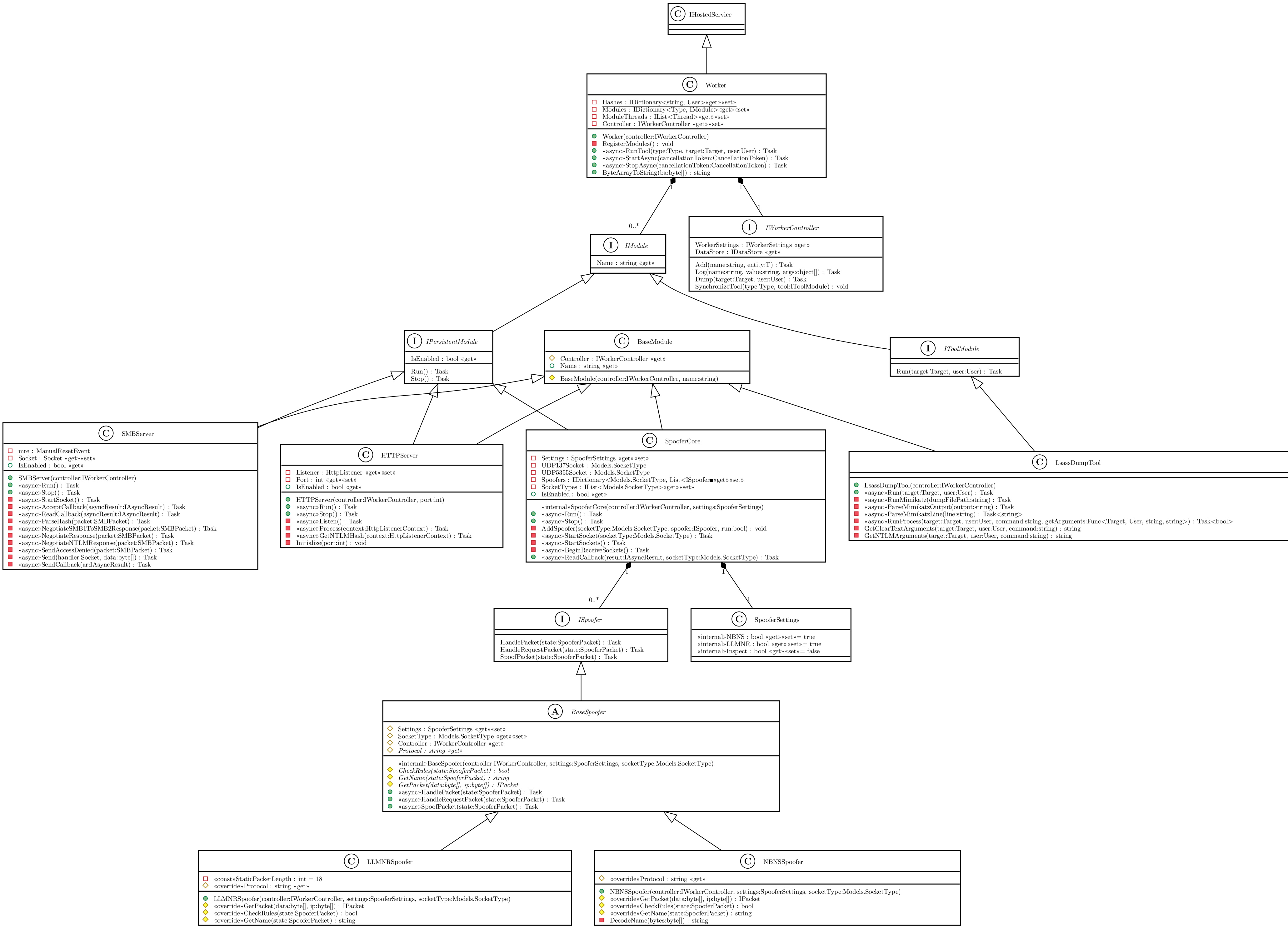
[39] Evan You. *What is Vuex?* URL: https://vuex.vuejs.org/ (visited on 01/16/2018).

[40] 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).

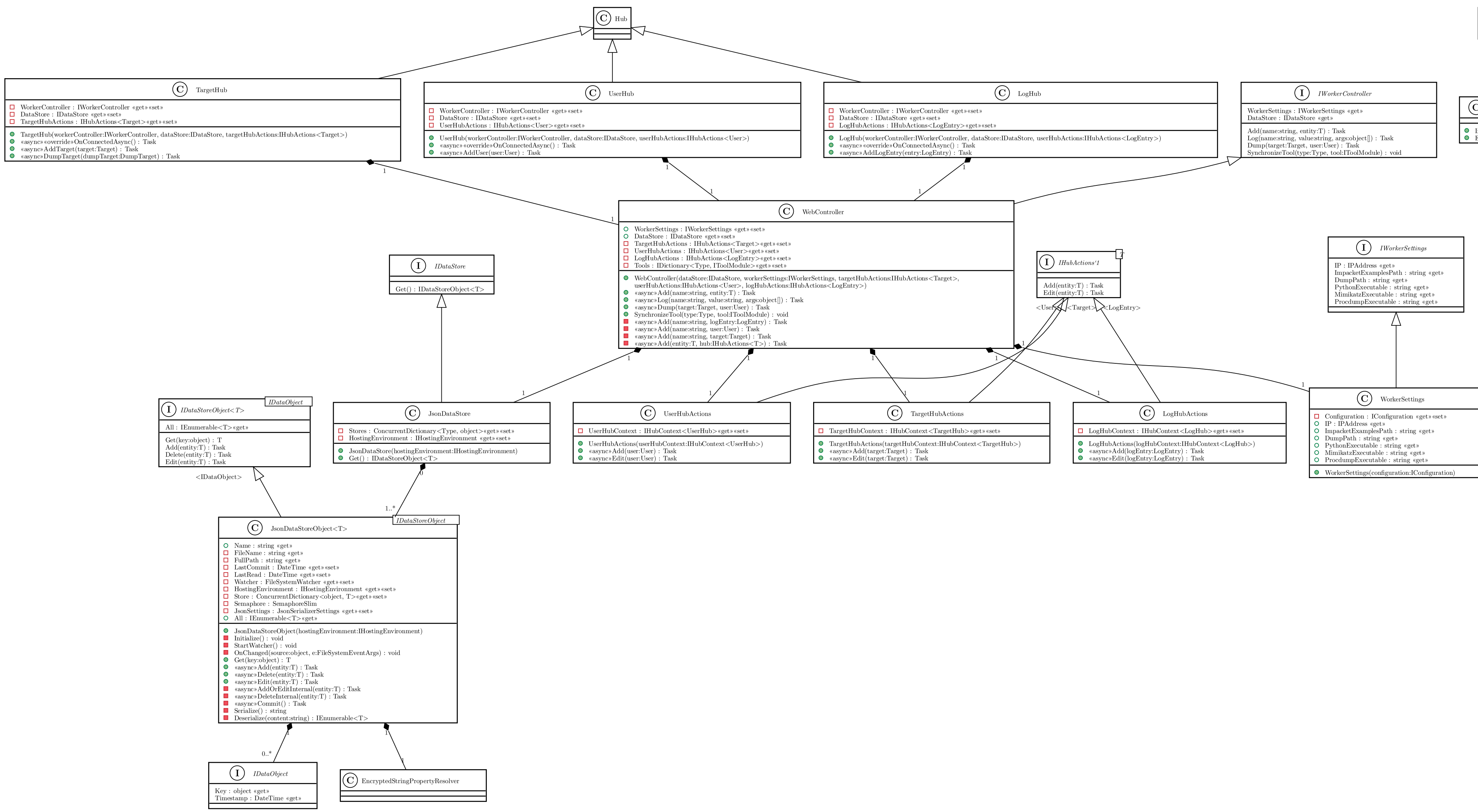
Appendices

A Class Diagrams

A.1 Worker class diagram


IHostedService
Worker
Hashes : IDictionary<string, User>«get»«set»
Modules : IDictionary<Type, IModule>«get»«set»
ModuleThreads : IList<Thread>«get»«set»
Controller : IWorkerController «get»«set»
Worker(controller:IWorkerController)
RegisterModules() : void
«async»RunTool(type:Type, target:Target, user:User) : Task
«async»StartAsync(cancellationToken:CancellationToken) : Task
«async»StopAsync(cancellationToken:CancellationToken) : Task
ByteArrayToString(ba:byte[]) : string
1
0..*
1
1
IModule
Name : string «get»
IWorkerController
WorkerSettings : IWorkerSettings «get»
DataStore : IDataStore «get»
Add(name:string, entity:T) : Task
Log(name:string, value:string, args:object[]) : Task
Dump(target:Target, user:User) : Task
SynchronizeTool(type:Type, tool:IToolModule) : void
IPersistentModule
IsEnabled : bool «get»
Run() : Task
Stop() : Task
BaseModule
Controller : IWorkerController «get»
Name : string «get»
BaseModule(controller:IWorkerController, name:string)
IToolModule
Run(target:Target, user:User) : Task
SMBServer
mre : ManualResetEvent
Socket : Socket «get»«set»
IsEnabled : bool «get»
SMBServer(controller:IWorkerController)
«async»Run() : Task
«async»Stop() : Task
«async»StartSocket() : Task
«async»AcceptCallback(asyncResult:IAsyncResult) : Task
«async»ReadCallback(asyncResult:IAsyncResult) : Task
«async»ParseHash(packet:SMBPacket) : Task
«async»NegotiateSMB1ToSMB2Response(packet:SMBPacket) : Task
«async»NegotiateResponse(packet:SMBPacket) : Task
«async»NegotiateNTLMResponse(packet:SMBPacket) : Task
«async»SendAccessDenied(packet:SMBPacket) : Task
«async»Send(handler:Socket, data:byte[]) : Task
«async»SendCallback(ar:IAsyncResult) : Task
HTTPServer
Listener : HttpListener «get»«set»
Port : int «get»«set»
IsEnabled : bool «get»
HTTPServer(controller:IWorkerController, port:int)
«async»Run() : Task
«async»Stop() : Task
«async»Listen() : Task
«async»Process(context:HttpListenerContext) : Task
«async»GetNTLMHash(context:HttpListenerContext) : Task
Initialize(port:int) : void
SpooferCore
Settings : SpooferSettings «get»«set»
UDP137Socket : Models.SocketType
UDP5355Socket : Models.SocketType
Spoofers : IDictionary<Models.SocketType, List<ISpoofer«get»«set»
SocketTypes : IList<Models.SocketType>«get»«set»
IsEnabled : bool «get»
«internal»SpooferCore(controller:IWorkerController, settings:SpooferSettings)
«async»Run() : Task
«async»Stop() : Task
AddSpoofer(socketType:Models.SocketType, spoofer:ISpoofer, run:bool) : void
«async»StartSocket(socketType:Models.SocketType) : Task
«async»StartSockets() : Task
«async»BeginReceiveSockets() : Task
«async»ReadCallback(result:IAsyncResult, socketType:Models.SocketType) : Task
LsassDumpTool
LsassDumpTool(controller:IWorkerController)
«async»Run(target:Target, user:User) : Task
«async»RunMimikatz(dumpFilePath:string) : Task
«async»ParseMimikatzOutput(output:string) : Task
«async»ParseMimikatzLine(line:string) : Task<string>
«async»RunProcess(target:Target, user:User, command:string, getArguments:Func<Target, User, string, string>) : Task<bool>
GetClearTextArguments(target:Target, user:User, command:string) : string
GetNTLMArguments(target:Target, user:User, command:string) : string
1
0..*
1
1
ISpoofer
HandlePacket(state:SpooferPacket) : Task
HandleRequestPacket(state:SpooferPacket) : Task
SpoofPacket(state:SpooferPacket) : Task
SpooferSettings
«internal»NBNS : bool «get»«set»= true
«internal»LLMNR : bool «get»«set»= true
«internal»Inspect : bool «get»«set»= false
BaseSpoofer
Settings : SpooferSettings «get»«set»
SocketType : Models.SocketType «get»«set»
Controller : IWorkerController «get»
Protocol : string «get»
«internal»BaseSpoofer(controller:IWorkerController, settings:SpooferSettings, socketType:Models.SocketType)
CheckRules(state:SpooferPacket) : bool
GetName(state:SpooferPacket) : string
GetPacket(data:byte[], ip:byte[]) : IPacket
«async»HandlePacket(state:SpooferPacket) : Task
«async»HandleRequestPacket(state:SpooferPacket) : Task
«async»SpoofPacket(state:SpooferPacket) : Task
LLMNRSpoofer
«const»StaticPacketLength : int = 18
«override»Protocol : string «get»
LLMNRSpoofer(controller:IWorkerController, settings:SpooferSettings, socketType:Models.SocketType)
«override»GetPacket(data:byte[], ip:byte[]) : IPacket
«override»CheckRules(state:SpooferPacket) : bool
«override»GetName(state:SpooferPacket) : string
NBNSSpoofer
«override»Protocol : string «get»
NBNSSpoofer(controller:IWorkerController, settings:SpooferSettings, socketType:Models.SocketType)
«override»GetPacket(data:byte[], ip:byte[]) : IPacket
«override»CheckRules(state:SpooferPacket) : bool
«override»GetName(state:SpooferPacket) : string
DecodeName(bytes:byte[]) : string

A.2 Web class diagram


Hub
Controller
TargetHub
WorkerController : IWorkerController «get»«set»
DataStore : IDataStore «get»«set»
TargetHubActions : IHubActions<Target>«get»«set»
TargetHub(workerController:IWorkerController, dataStore:IDataStore, targetHubActions:IHubActions<Target>)
«async»«override»OnConnectedAsync() : Task
«async»AddTarget(target:Target) : Task
«async»DumpTarget(dumpTarget:DumpTarget) : Task
UserHub
WorkerController : IWorkerController «get»«set»
DataStore : IDataStore «get»«set»
UserHubActions : IHubActions<User>«get»«set»
UserHub(workerController:IWorkerController, dataStore:IDataStore, userHubActions:IHubActions<User>)
«async»«override»OnConnectedAsync() : Task
«async»AddUser(user:User) : Task
LogHub
WorkerController : IWorkerController «get»«set»
DataStore : IDataStore «get»«set»
LogHubActions : IHubActions<LogEntry>«get»«set»
LogHub(workerController:IWorkerController, dataStore:IDataStore, userHubActions:IHubActions<LogEntry>)
«async»«override»OnConnectedAsync() : Task
«async»AddLogEntry(entry:LogEntry) : Task
IWorkerController
WorkerSettings : IWorkerSettings «get»
DataStore : IDataStore «get»
Add(name:string, entity:T) : Task
Log(name:string, value:string, args:object[]) : Task
Dump(target:Target, user:User) : Task
SynchronizeTool(type:Type, tool:IToolModule) : void
HomeController
Index() : IActionResult
Error() : IActionResult
WebController
WorkerSettings : IWorkerSettings «get»«set»
DataStore : IDataStore «get»«set»
TargetHubActions : IHubActions<Target>«get»«set»
UserHubActions : IHubActions<User>«get»«set»
LogHubActions : IHubActions<LogEntry>«get»«set»
Tools : IDictionary<Type, IToolModule>«get»«set»
WebController(dataStore:IDataStore, workerSettings:IWorkerSettings, targetHubActions:IHubActions<Target>,
userHubActions:IHubActions<User>, logHubActions:IHubActions<LogEntry>)
«async»Add(name:string, entity:T) : Task
«async»Log(name:string, value:string, args:object[]) : Task
«async»Dump(target:Target, user:User) : Task
SynchronizeTool(type:Type, tool:IToolModule) : void
«async»Add(name:string, logEntry:LogEntry) : Task
«async»Add(name:string, user:User) : Task
«async»Add(name:string, target:Target) : Task
«async»Add(entity:T, hub:IHubActions<T>) : Task
IDataStore
Get() : IDataStoreObject<T>
IHubActions'1
T
Add(entity:T) : Task
Edit(entity:T) : Task
<User>
<Target>
<LogEntry>
IWorkerSettings
IP : IPAddress «get»
ImpacketExamplesPath : string «get»
DumpPath : string «get»
PythonExecutable : string «get»
MimikatzExecutable : string «get»
ProcdumpExecutable : string «get»
IDataStoreObject<T>
IDataObject
All : IEnumerable<T>«get»
Get(key:object) : T
Add(entity:T) : Task
Delete(entity:T) : Task
Edit(entity:T) : Task
<IDataObject>
JsonDataStore
Stores : ConcurrentDictionary<Type, object>«get»«set»
HostingEnvironment : IHostingEnvironment «get»«set»
JsonDataStore(hostingEnvironment:IHostingEnvironment)
Get() : IDataStoreObject<T>
UserHubActions
UserHubContext : IHubContext<UserHub>«get»«set»
UserHubActions(userHubContext:IHubContext<UserHub>)
«async»Add(user:User) : Task
«async»Edit(user:User) : Task
TargetHubActions
TargetHubContext : IHubContext<TargetHub>«get»«set»
TargetHubActions(targetHubContext:IHubContext<TargetHub>)
«async»Add(target:Target) : Task
«async»Edit(target:Target) : Task
LogHubActions
LogHubContext : IHubContext<LogHub>«get»«set»
LogHubActions(logHubContext:IHubContext<LogHub>)
«async»Add(logEntry:LogEntry) : Task
«async»Edit(logEntry:LogEntry) : Task
WorkerSettings
Configuration : IConfiguration «get»«set»
IP : IPAddress «get»
ImpacketExamplesPath : string «get»
DumpPath : string «get»
PythonExecutable : string «get»
MimikatzExecutable : string «get»
ProcdumpExecutable : string «get»
WorkerSettings(configuration:IConfiguration)
JsonDataStoreObject<T>
IDataStoreObject
Name : string «get»
FileName : string «get»
FullPath : string «get»
LastCommit : DateTime «get»«set»
LastRead : DateTime «get»«set»
Watcher : FileSystemWatcher «get»«set»
HostingEnvironment : IHostingEnvironment «get»«set»
Store : ConcurrentDictionary<object, T>«get»«set»
Semaphore : SemaphoreSlim
JsonSettings : JsonSerializerSettings «get»«set»
All : IEnumerable<T>«get»
JsonDataStoreObject(hostingEnvironment:IHostingEnvironment)
Initialize() : void
StartWatcher() : void
OnChanged(source:object, e:FileSystemEventArgs) : void
Get(key:object) : T
«async»Add(entity:T) : Task
«async»Delete(entity:T) : Task
«async»Edit(entity:T) : Task
«async»AddOrEditInternal(entity:T) : Task
«async»DeleteInternal(entity:T) : Task
«async»Commit() : Task
Serialize() : string
Deserialize(content:string) : IEnumerable<T>
IDataObject
Key : object «get»
Timestamp : DateTime «get»
EncryptedStringPropertyResolver

A.3 Data objects class diagram