quickbreach@defcon26:~# ./smbetray.py --help

Backdooring & Breaking Signatures

William Martin (@QuickBreach)

> whoami

- William Martin
- OSCP
- Penetration Tester
- Supervisor at RSM US LLP in Charlotte, NC
- Second time presenting at DEFCON

- Twitter: @QuickBreach

> Who is this talk for?

- Red teamers looking to learn more about Active Directory, SMB security, and pick up new attacks against insecure SMB connections
- Blue teamers that want to stop the red teamers from using what they learn
- Anyone curious about how SMB signing actually works

> Overview

- Brief recap on what SMB is
- NTLMv2 Relay attack
- Investigate what SMB signing actually is
- How else we can attack SMB?
- Introduce SMBetray
- Demo & tool release
- Countermeasures
- Credits

Recap on SMB

> Terminology clarification

SMB server = Any PC receiving the SMB connection, ***not*** necessarily a Windows Server OS. Eg, a Windows 7 box can be the SMB server, as every Windows OS runs an SMB server by default

SMB client = The PC/Server connecting to the SMB server

> Recap on SMB

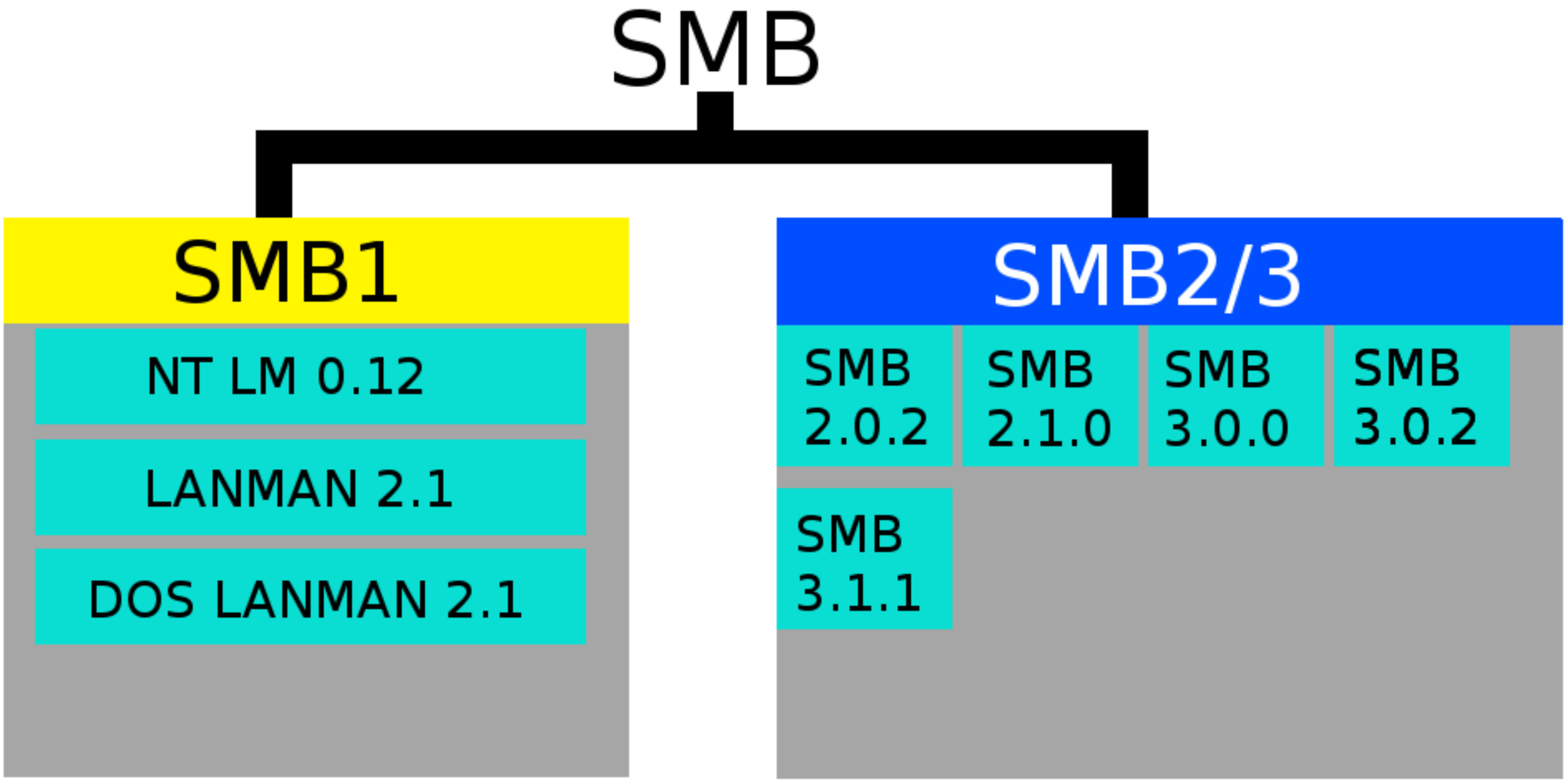
The Server Message Block (SMB) Protocol is a network file sharing protocol, and as implemented in Microsoft Windows is known as Microsoft SMB Protocol. The set of message packets that defines a particular version of the protocol is called a dialect. The Common Internet File System (CIFS) Protocol is a dialect of SMB. Both SMB and CIFS are also available on VMS, several versions of Unix, and other operating systems.

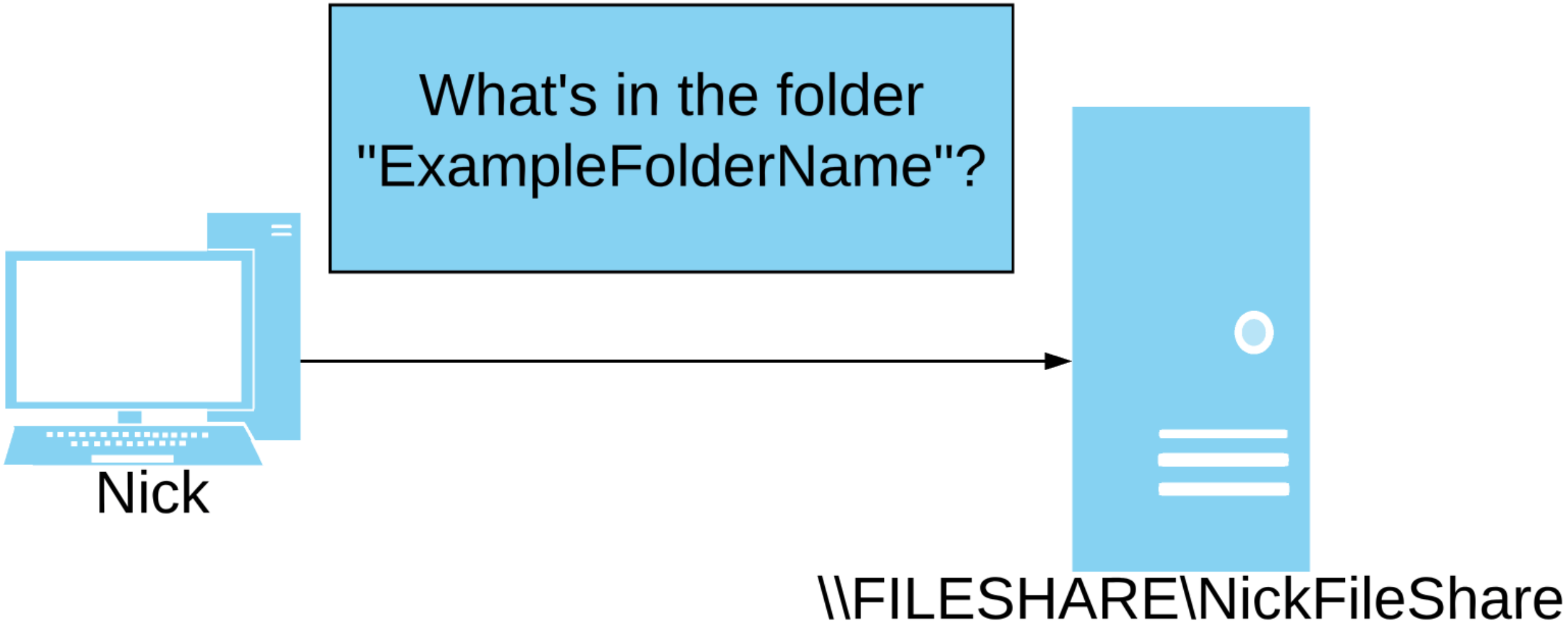
(Source: https://docs.microsoft.com/en-us/windows/desktop/fileio/microsoft-smb-protocol-and-cifs-protocol-overview)

> Recap on SMB

SMB listens on TCP port 445 and allows for file sharing and management over the network, with features including:

- Mapping network drives
- Reading & writing files to shares
- Authentication support
- Providing access to MSRPC named pipes


SMB
SMB1
NT LM 0.12
LANMAN 2.1
DOS LANMAN 2.1
SMB2/3
SMB 2.0.2
SMB 2.1.0
SMB 3.0.0
SMB 3.0.2
SMB 3.1.1


What's in the folder "ExampleFolderName"?
Nick
\\FILESHARE\NickFileShare

A_Document.docx
Desktop.ini
TotallyNotPasswords.xlsx

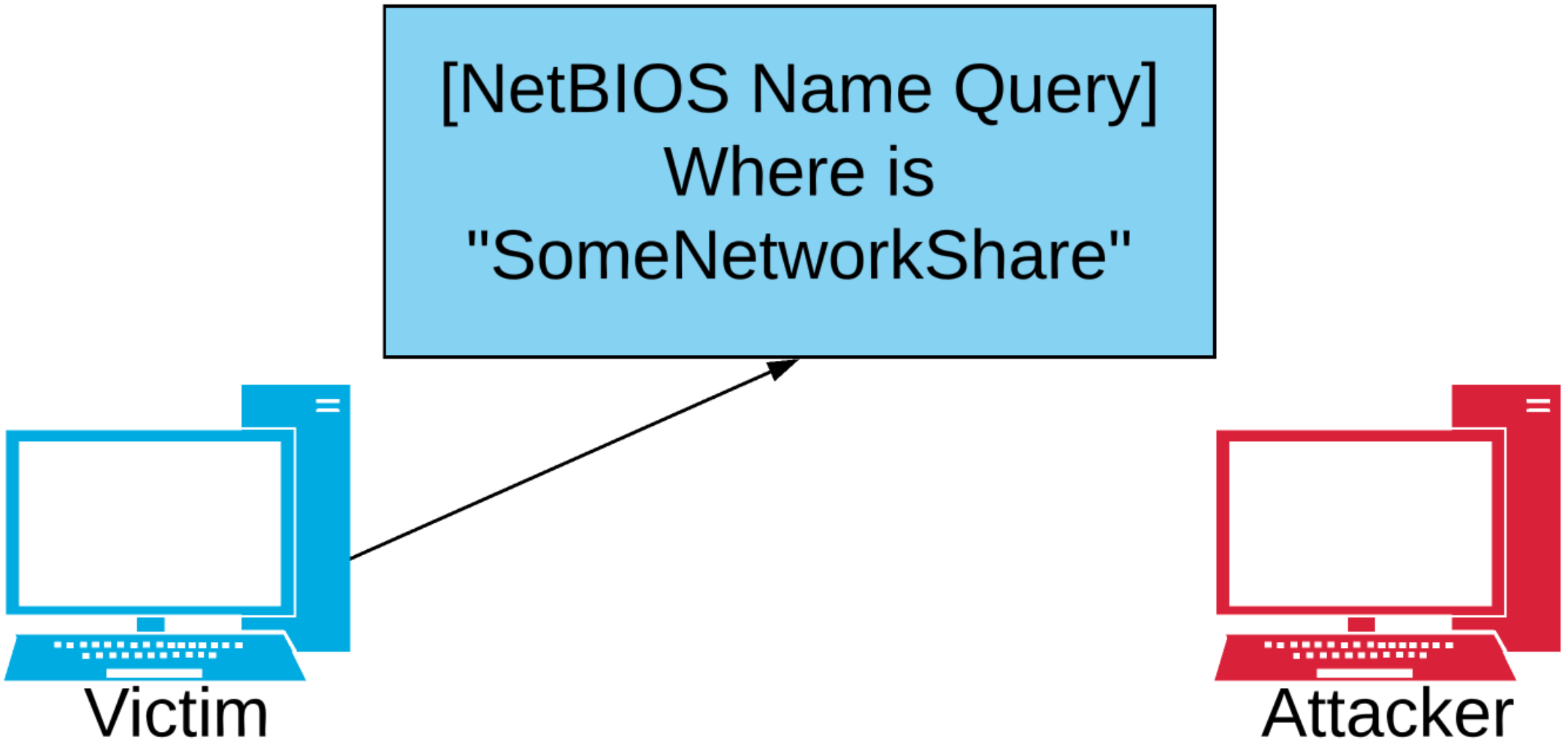
> Recap on SMB

Attackers love it for:

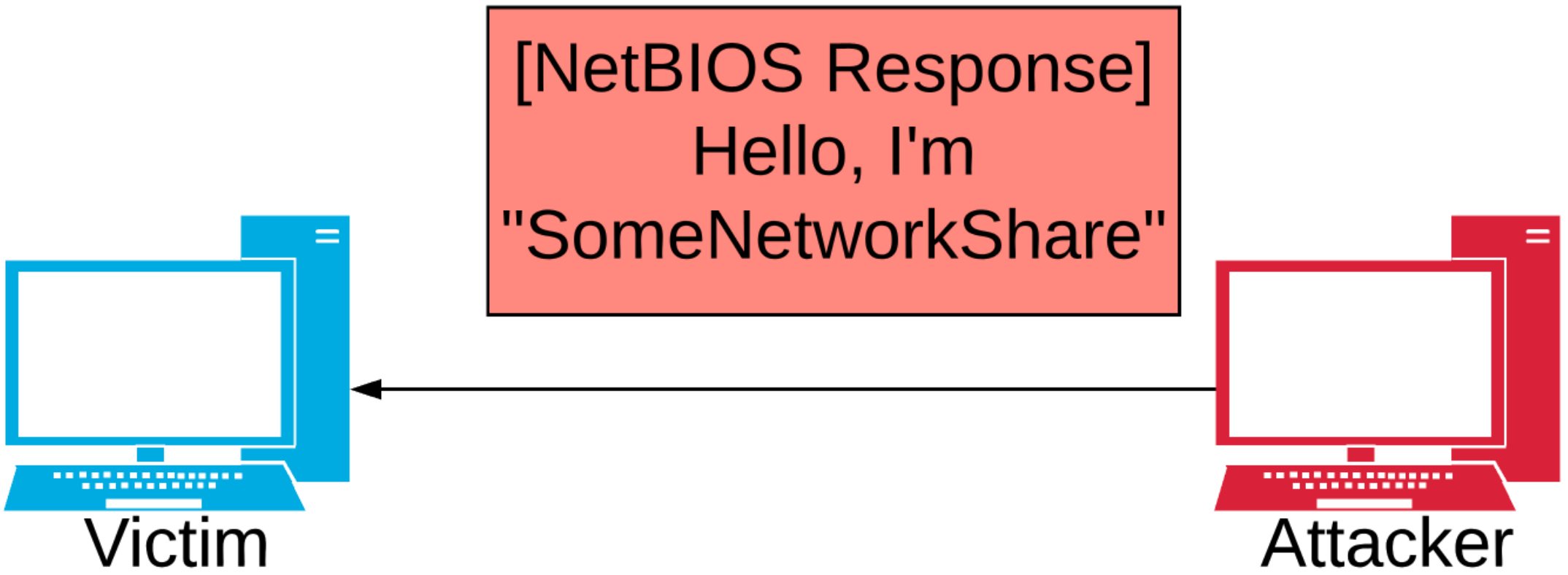
- Pass-the-hash
- System enumeration (authenticated, or null sessions)
- Spidering shares & hunting for sensitive data, such as for the cpassword key in SYSVOL xml files


MS17-010
MS08-067
imgflip.com

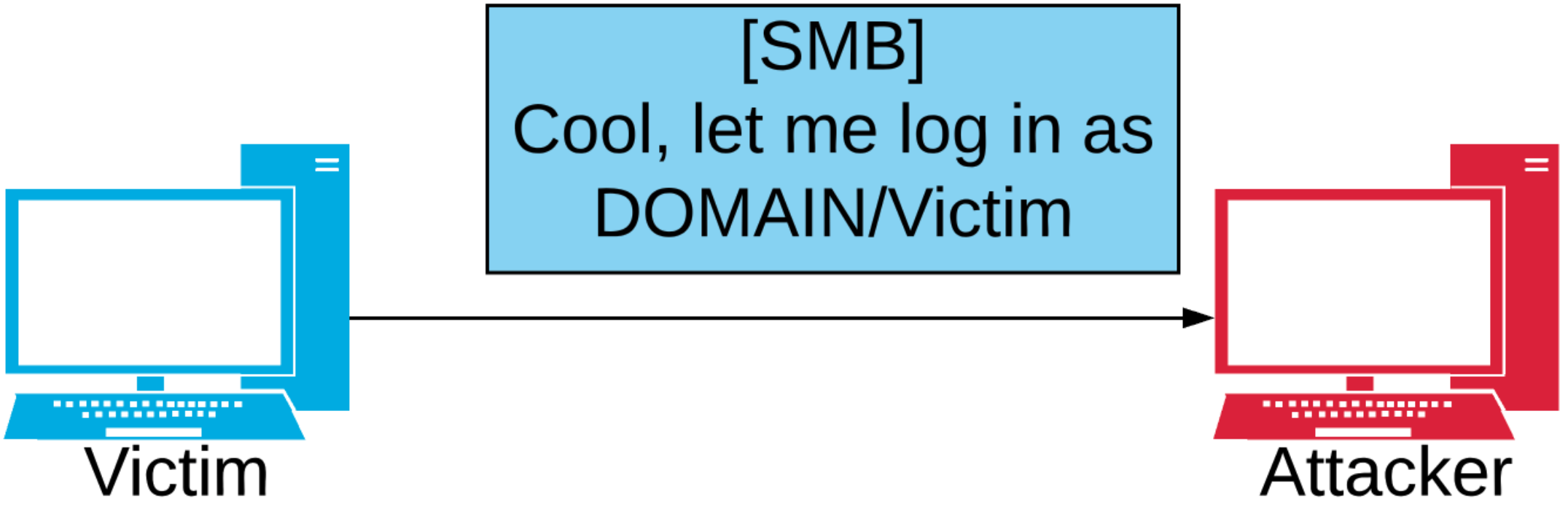
LLMNR & NBT-NS NTLM Relay



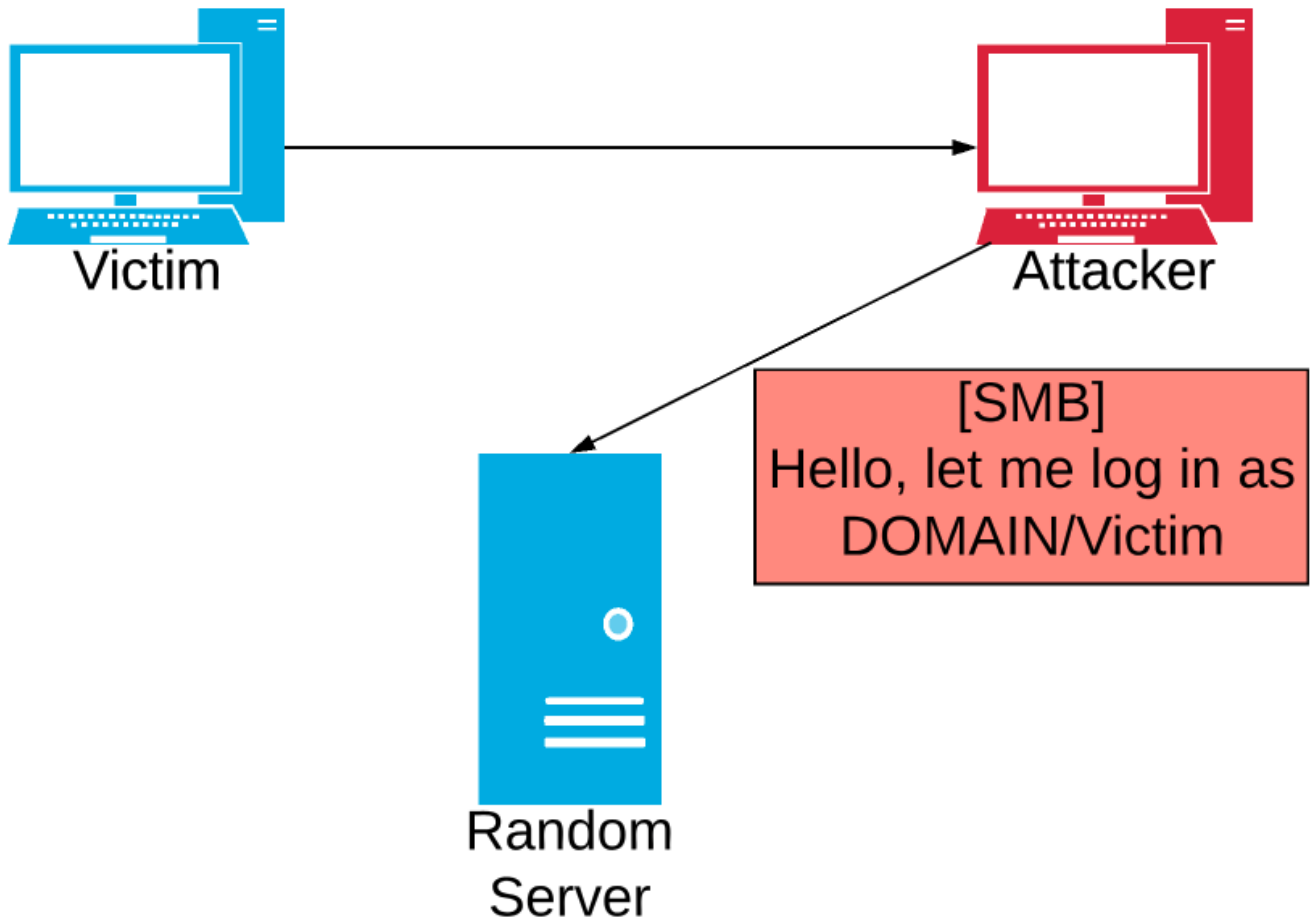
LLMNR & NBT-NS NTLM Relay



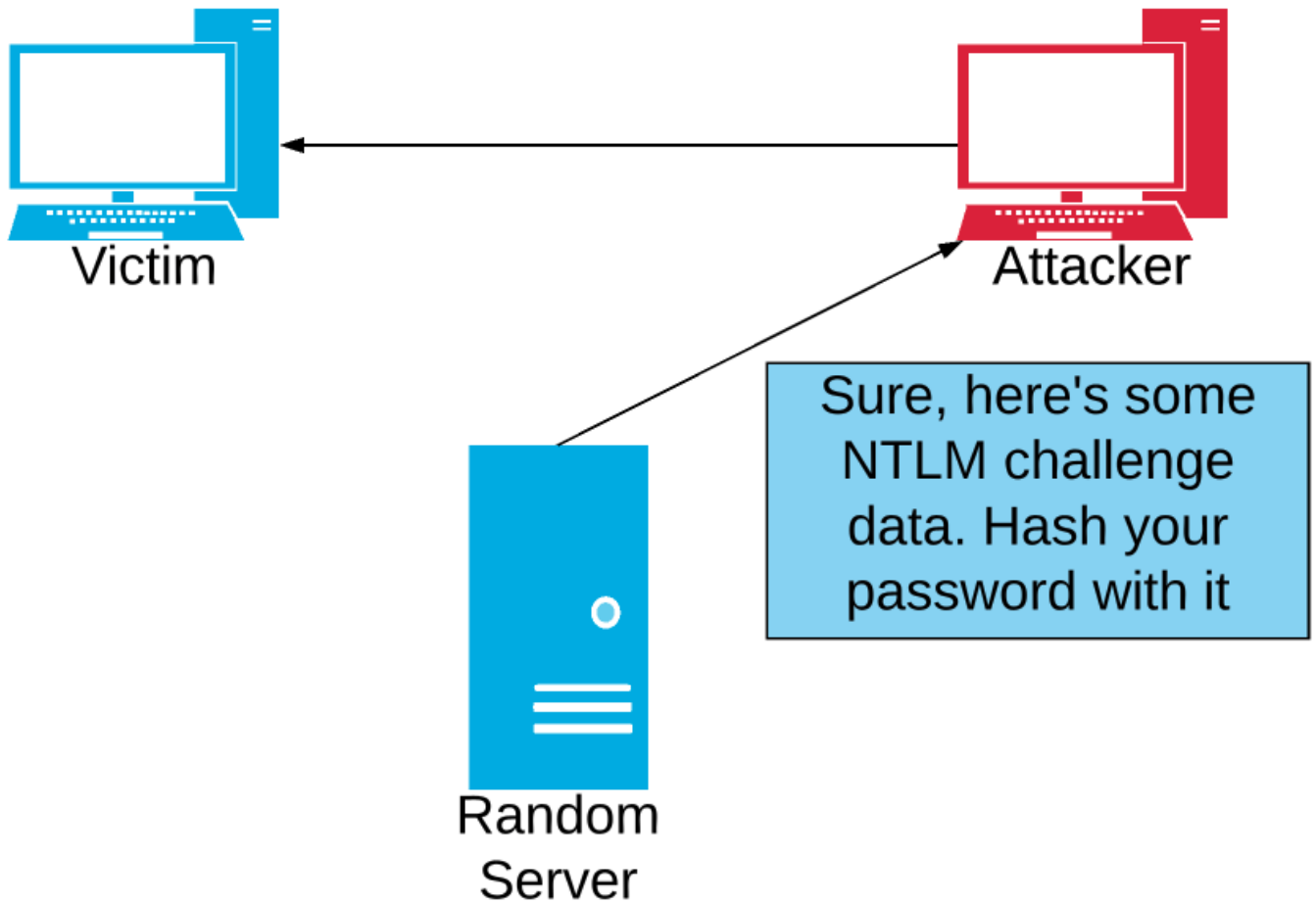
LLMNR & NBT-NS NTLM Relay



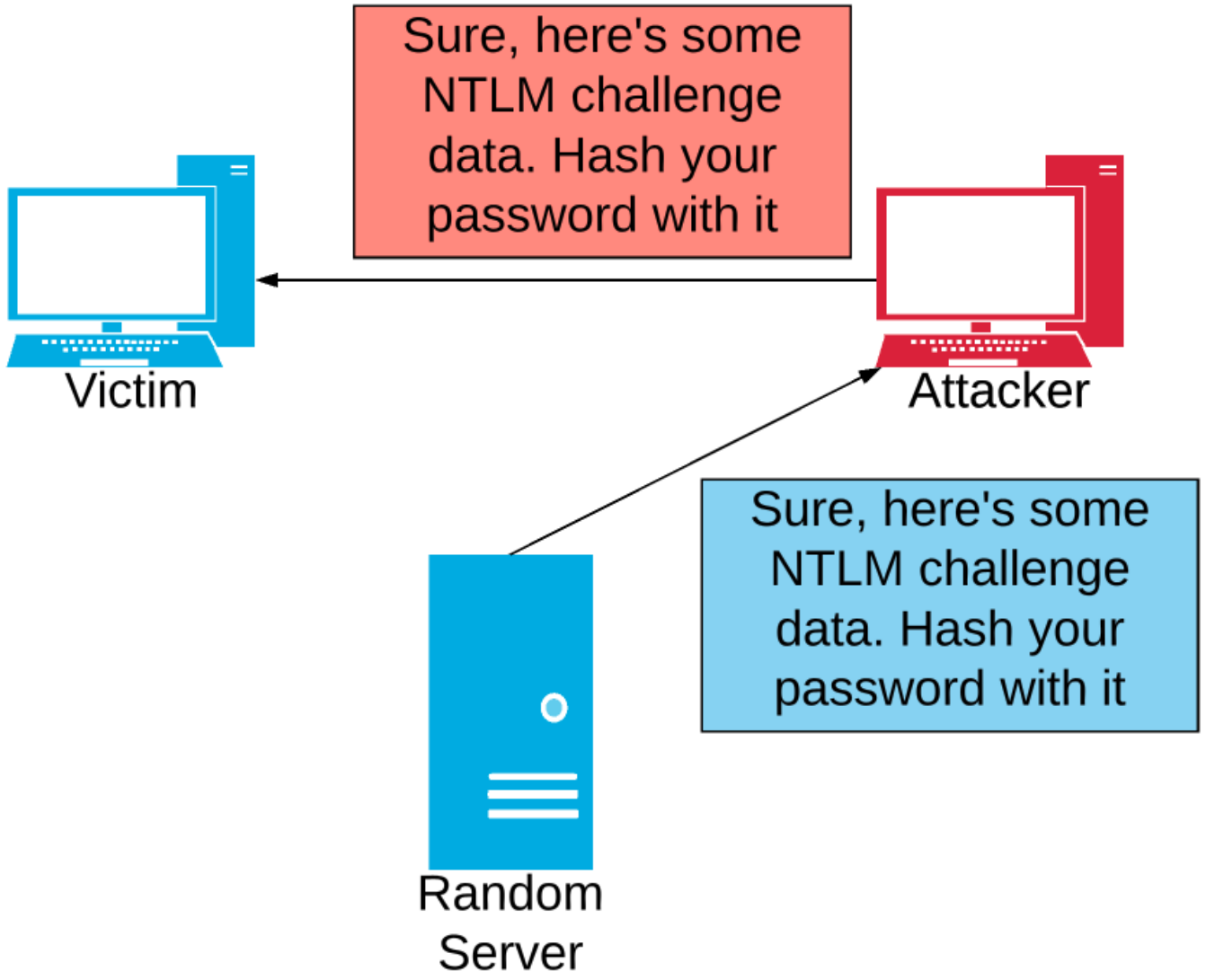
LLMNR & NBT-NS NTLM Relay



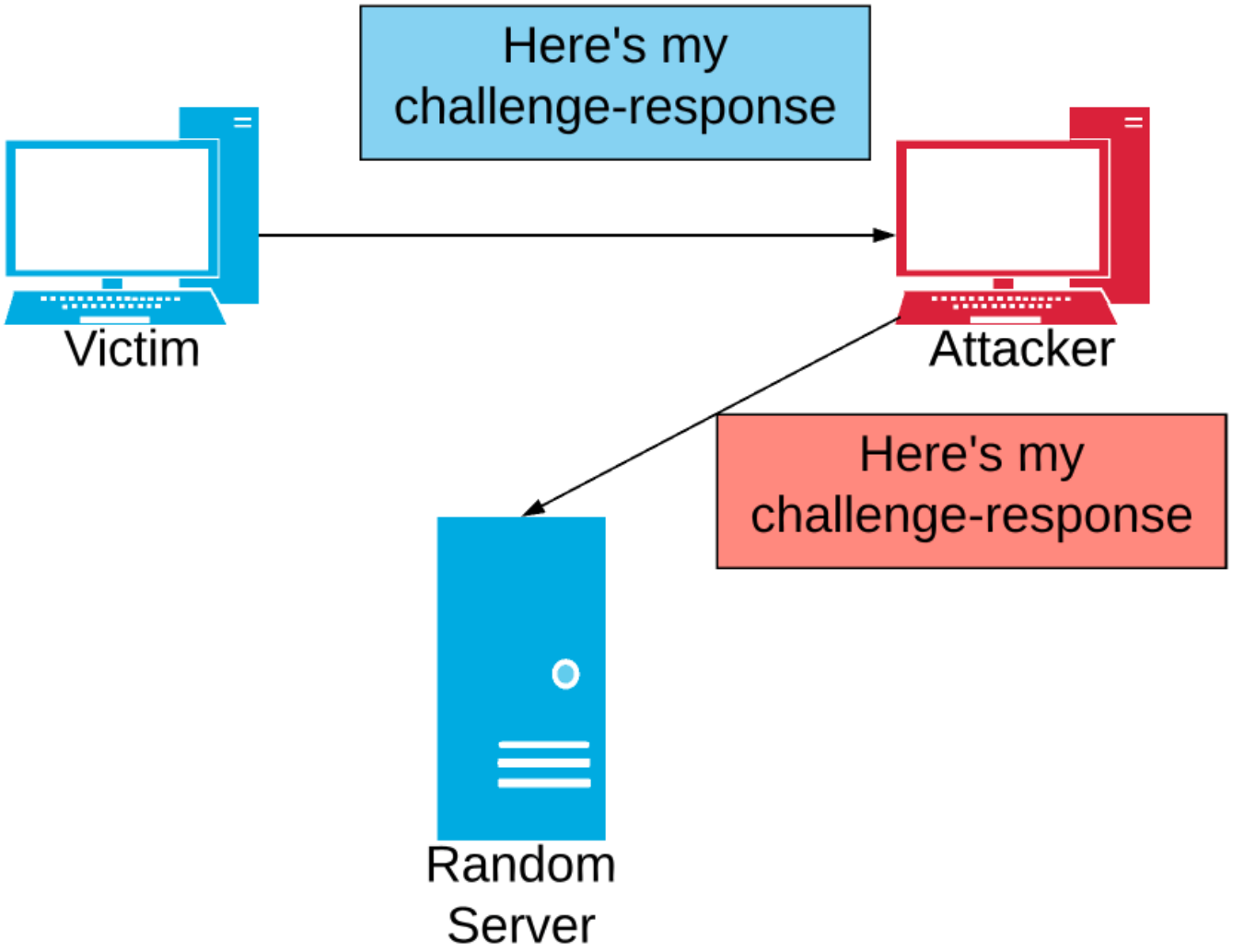
LLMNR & NBT-NS NTLM Relay



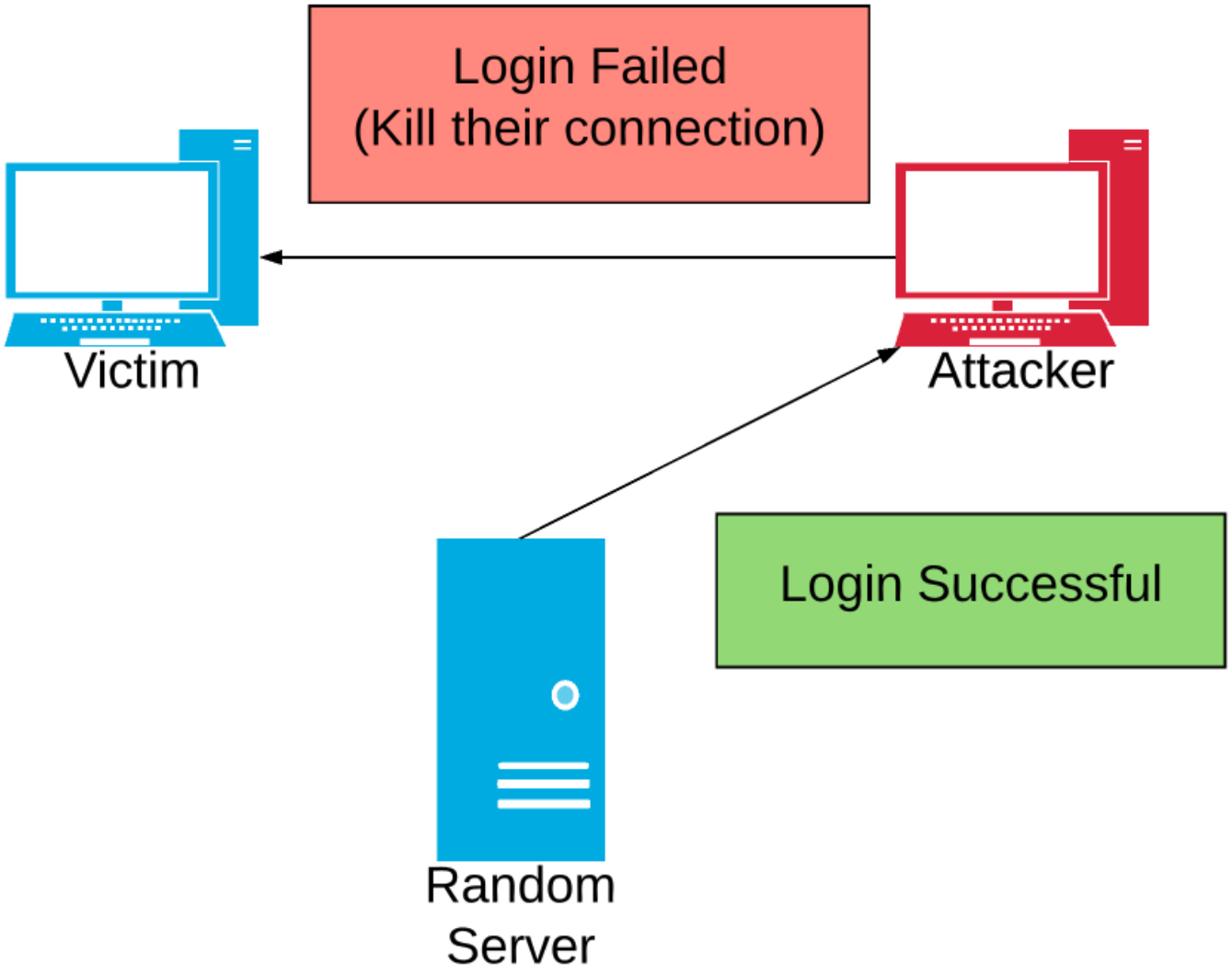
LLMNR & NBT-NS NTLM Relay



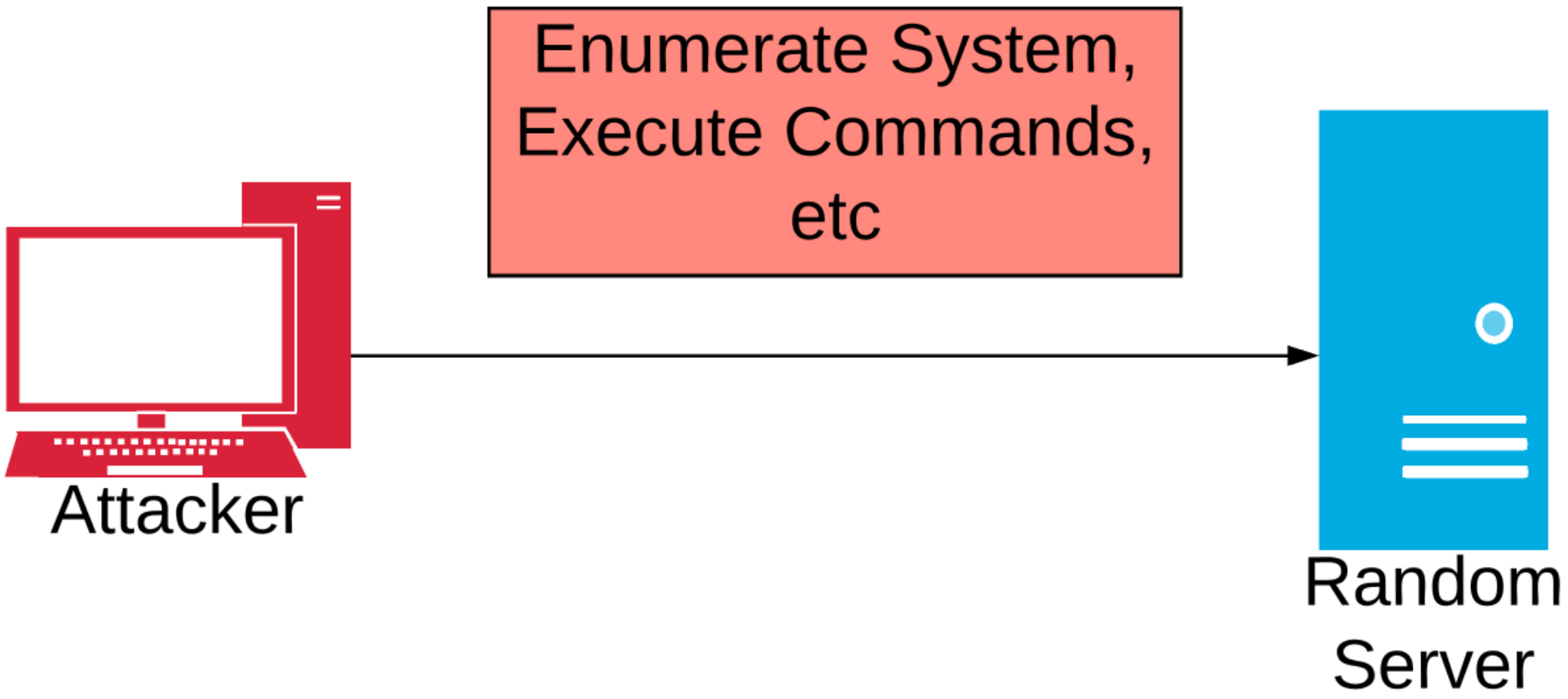
LLMNR & NBT-NS NTLM Relay



LLMNR & NBT-NS NTLM Relay



LLMNR & NBT-NS NTLM Relay



```
Impacket v0.9.18-dev - Copyright 2002-2018 Core Security Technologies

[*] Running in relay mode
[*] Setting up SMB Server
[*] Setting up HTTP Server

[*] Servers started, waiting for connections
[*] SMBD: Received connection from 10.0.0.3, attacking target 10.0.0.250
[-]  Signature is REQUIRED on the other end, attack will not work
[-] Authenticating against 10.0.0.250 as \demo FAILED
```

What is SMB signing?

> What is SMB signing?

MEDIUM SMB Signing not required

Description

Signing is not required on the remote SMB server. An unauthenticated, remote attacker can exploit this to conduct man-in-the-middle attacks against the SMB server.

Solution

Enforce message signing in the host's configuration. On Windows, this is found in the policy setting 'Microsoft network server: Digitally sign communications (always)'. On Samba, the setting is called 'server signing'. See the 'see also' links for further details.

> What is SMB signing?

What is SMB Signing?

Server Message Block (SMB) signing is a Microsoft-devised security mechanism that attempts to prevent man-in-the-middle attacks. If a network administrator configures SMB signing on clients and servers, signatures are added to the packet header. A decrypted signature by the recipient server or client indicates a valid packet. If the signature is malformed or not present, or if the SMB packet is compromised, the client or server rejects and drops the packet.

> What is SMB signing?

Server Message Block (SMB) is the file protocol most commonly used by Windows. SMB Signing is a feature through which communications using SMB can be digitally signed at the packet level. Digitally signing the packets enables the recipient of the packets to confirm their point of origination and their authenticity. This security mechanism in the SMB protocol helps avoid issues like tampering of packets and “man in the middle” attacks.

Server Message Block security has two main components: user-level and share-level. The first is for accessing servers, and the second is for accessing files, folders, and printers if share-level authentication has been configured on the server. Most readers of this column already know about these aspects of SMB security, but you may not know about another feature called “SMB signing.” This is a feature that is available in all versions of Windows since NT4. It places a digital signature into each server message block, which is used by both SMB clients and servers to prevent so-called “man-in-the-middle” attacks and

> What is SMB signing?

Using SMB Packet Signing

Although SMB packet signing does not encrypt data, it does digitally sign Server Message Block (SMB) packets to ensure that the data has not been changed while in transit. The Management Server uses the Server Message Block (SMB) port (TCP/UDP 445) to deliver the files needed for agent installation on remote computers and for updating agent settings after installation.

You can configure this method by enabling the **Microsoft network client: Digitally sign communications (always)** and the **Microsoft network server: Digitally sign communications (always)** options. These options configure Windows 2000 to require the SMB server to perform SMB packet signing.

Enabling both of these options can mitigate "man-in-the-middle" attacks using SMB packets. These options can be configured using the Global or Local Policy snap-in for the MMC.

> What is SMB signing?

What I knew in the beginning:

- Protects the integrity of SMB messages between the client and server, preventing any tampering
- Is required by default on all domain controllers
- Occurs after authenticated session setup
- Stops my favorite attack


Diving into MSDN docs like

3.1.1.2 Cryptographic Material

Kerberos V5 establishes a secret key that is shared by a principal and the KDC and a session key that forms the basis for privacy between client and server. When KILE creates an
3.9). KILE concatenates the following information

- User accounts: < DNS of the realm, conve
- Computer accounts: < DNS name of the r
  DNS name of the realm, converted to lowe

Using KILE, application clients (for example, CIF
protocol MUST document the explicit use of the k

```
Define NTOWFv2(Passwd, User, UserDom
MD4(UNICODE(Passwd)), UNICODE(Concate
UserDom ) ) )
EndDefine

Define LMOWFv2(Passwd, User, UserDom
UserDom)
EndDefine

Set ResponseKeyNT to NTOWFv2(Passwd,
Set ResponseKeyLM to LMOWFv2(Passwd,

Define ComputeResponse(NegFlg, Respor
CHALLENGE_MESSAGE.ServerChallenge, Cl
As
If (User is set to "" && Passwd is se
    -- Special case for anonymous aut
    Set NtChallengeResponseLen to 0
    Set NtChallengeResponseMaxLen to 0
    Set NtChallengeResponseBufferOffset to 0
    Set LmChallengeResponse to Z(1)
Else
    Set temp to ConcatenationOf(Responserversion, HiResponserversion,
    Z(6), Time, ClientChallenge, Z(4), ServerName, Z(4))
    Set NTProofStr to HMAC_MD5(ResponseKeyNT,
    ConcatenationOf(CHALLENGE_MESSAGE.ServerChallenge,temp))
    Set NtChallengeResponse to ConcatenationOf(NTProofStr, temp)
    Set LmChallengeResponse to ConcatenationOf(HMAC_MD5(ResponseKeyLM,
    ConcatenationOf(CHALLENGE_MESSAGE.ServerChallenge, ClientChallenge))
    ClientChallenge )
EndIf

Set SessionBaseKey to HMAC_MD5(ResponseKeyNT, NTProofStr)
EndDefine
```

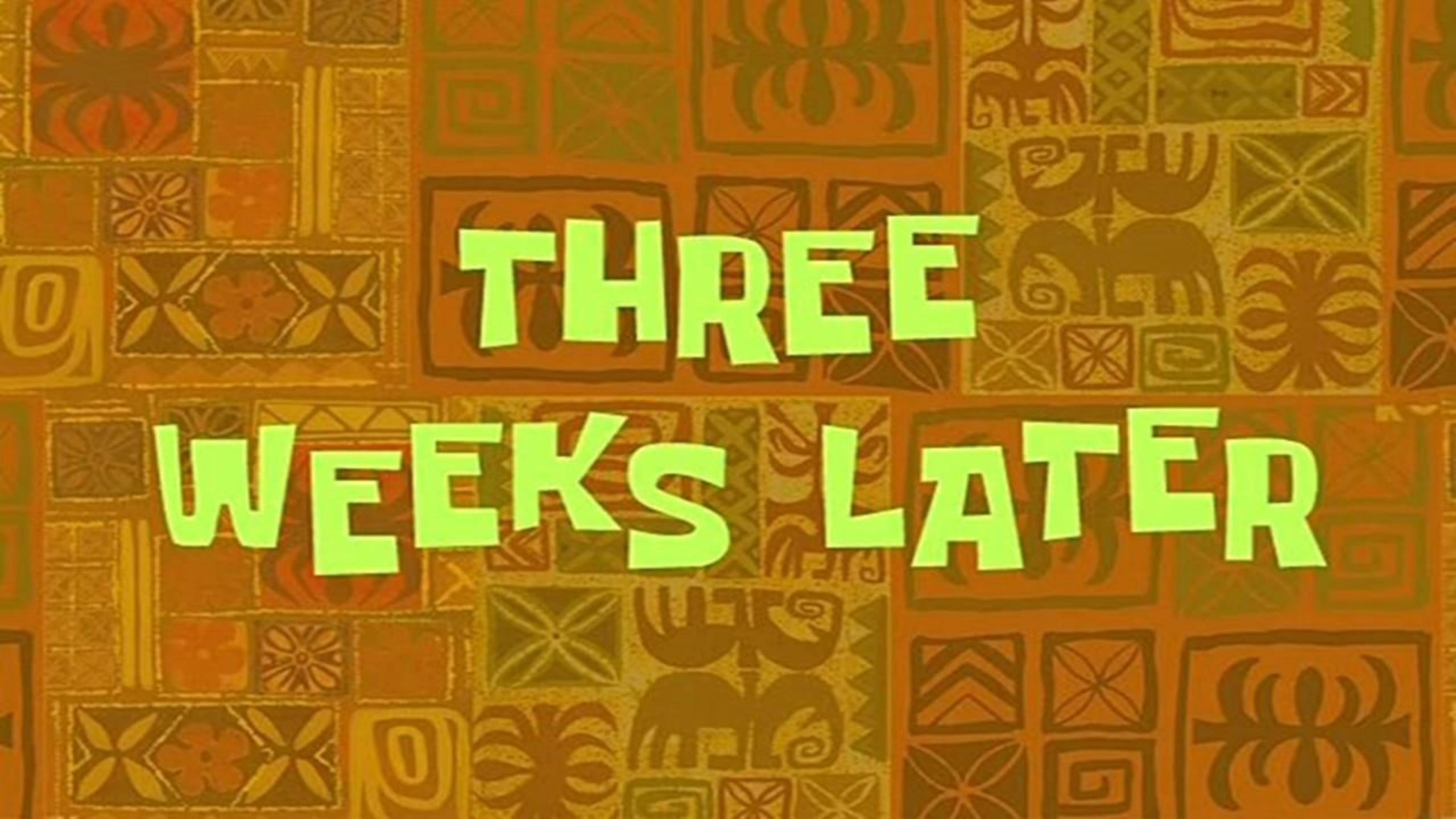
```
esponseServerVersion = '\x01'
iResponseServerVersion = '\x01'
esponseKeyNT = NTOWFv2(user, password, domain, nthash)
esponseKeyLM = LMOWFv2(user, password, domain, lmhash)

v_pairs = AV_PAIRS(serverName)
 In order to support SPN target name validation, we have to add this to the se
 get access denied
 This is set at Local Security Policy -> Local Policies -> Security Options ->
 level
f TEST_CASE is False:
   av_pairs[NTLMSSP_AV_TARGET_NAME] = 'cifs/'.encode('utf-16le') + av_pairs[NT
   if av_pairs[NTLMSSP_AV_TIME] is not None:
      aTime = av_pairs[NTLMSSP_AV_TIME][1]
   else:
      aTime = struct.pack('<q', (116444736000000000 + calendar.timegm(time.gmt
      av_pairs[NTLMSSP_AV_TIME] = aTime
   serverName = av_pairs.getData()
lse:
   aTime = '\x00'*8

emp = responseServerVersion + hiResponseServerVersion + '\x00' * 6 + aTime + c
      serverName + '\x00' * 4

tProofStr = hmac_md5(responseKeyNT, serverChallenge + temp)

tChallengeResponse = ntProofStr + temp
mChallengeResponse = hmac_md5(responseKeyNT, serverChallenge + clientChallenge
```


THREE
WEEKS LATER

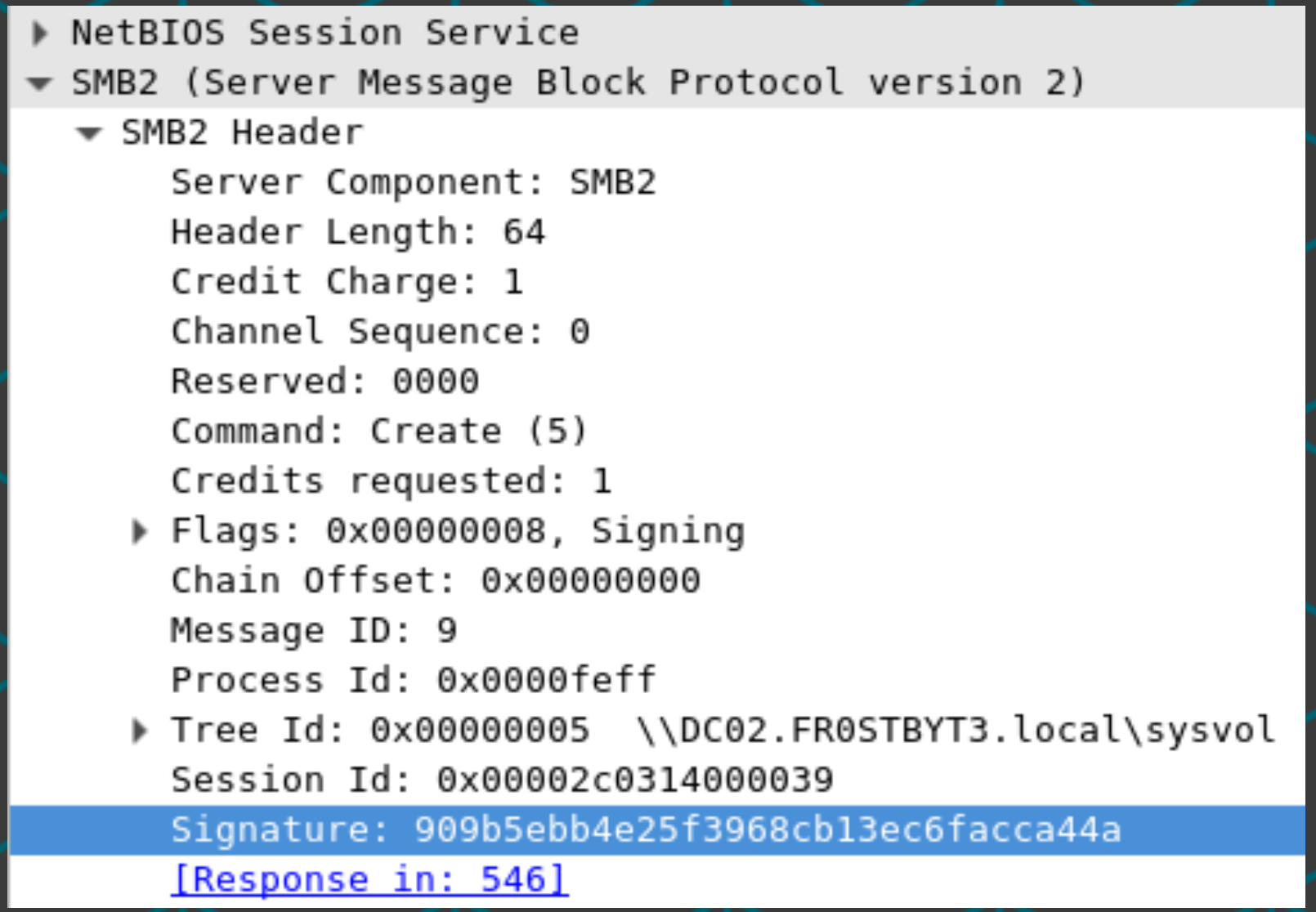
> What is SMB signing?

Lessons were learned – core concepts:

At the end of the authentication phase in an authenticated session, the client and server will possess the same 16-byte SessionBaseKey.

Depending on the dialect, the client and server may use this SessionBaseKey to generate subsequent keys each purposed for specific actions such as signing and encrypting SMB packets. Only those in possession of the keys can generate the appropriate signatures.

> Sample signature



> What is SMB signing?

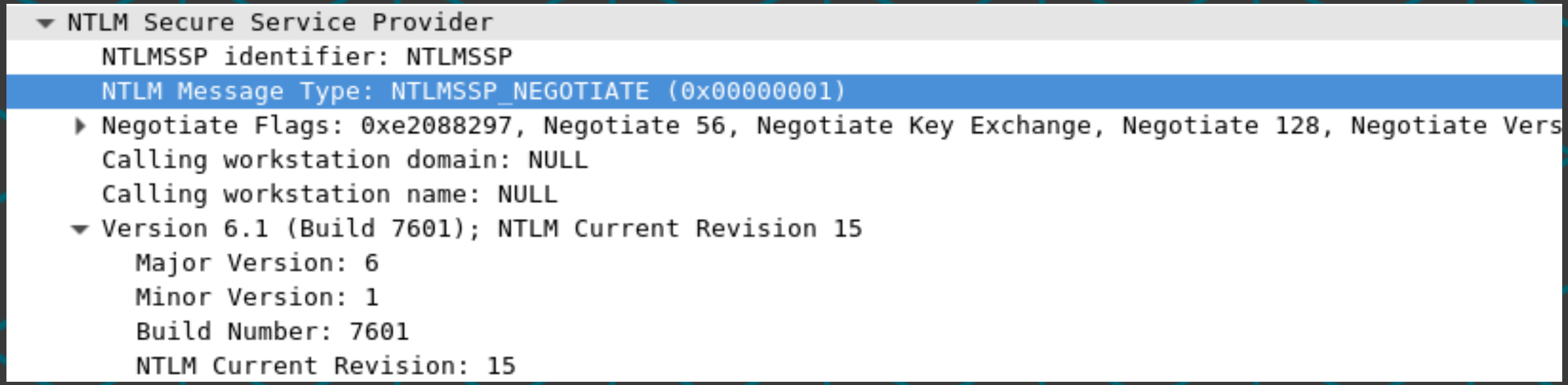
So, where does this SessionBaseKey come from – especially if we can manipulate the entire authentication process?

Answer: The creation of the SessionBaseKey depends on the authentication mechanism used.

> A walk through the NTLMv2 process

```
Setup Request, NTLMSSP_NEGOTIATE
Setup Response, Error: STATUS_MORE_PROCESSING_REQUIRED, NTLMSSP_CHALLENGE
Setup Request, NTLMSSP_AUTH, User: FR0STBYT3\CLIENT-PC$
```

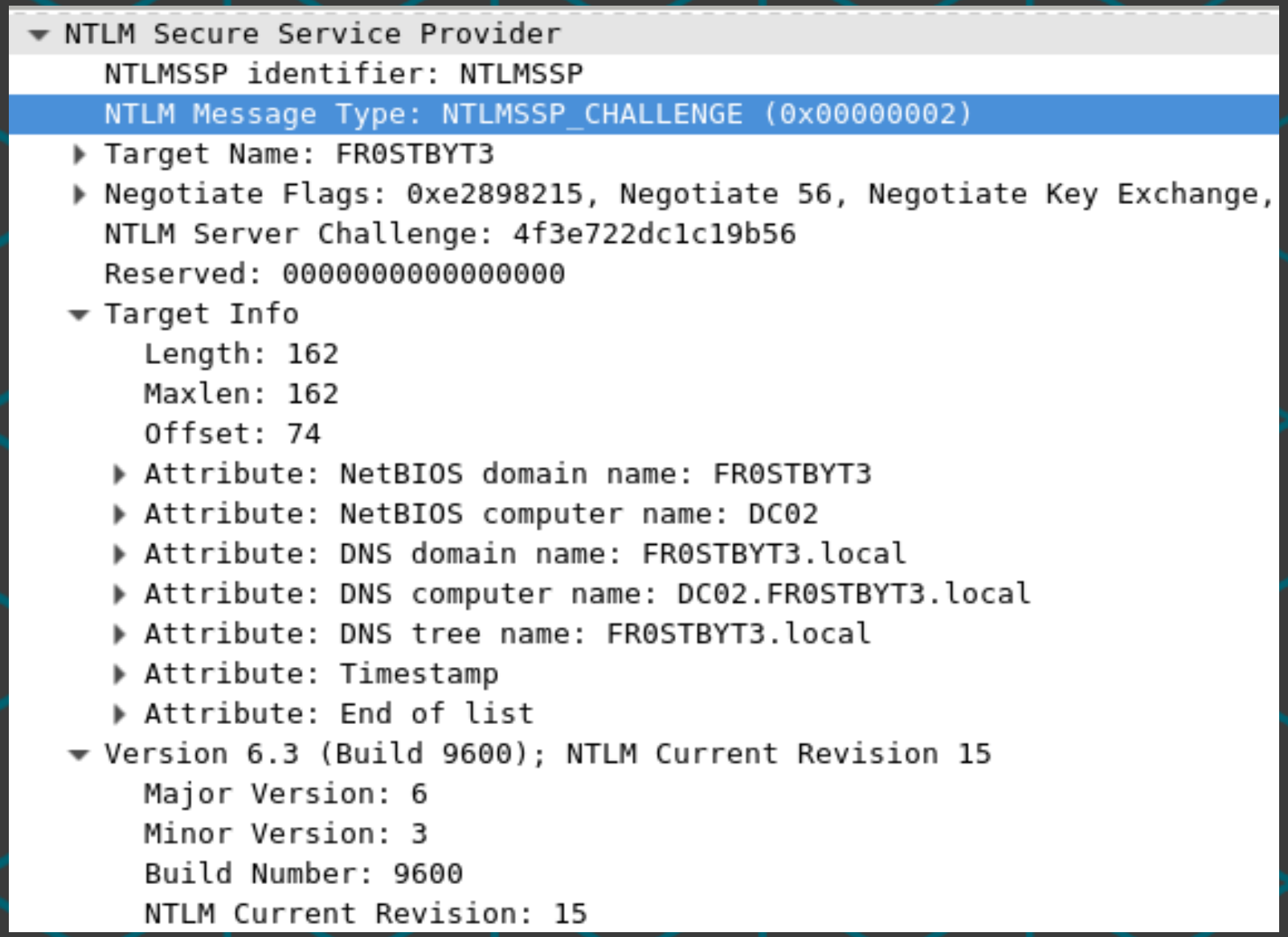
> NTLMSSP Negotiate



"Hello, I want to authenticate using NTLM"

- No username/password/domain information in message
- "Negotiate Key Exchange" is usually set. This means after the client authenticates and generates a SessionBaseKey, the client will generate a new random one, RC4 encrypt it with the old one, and send it to the server to use going forward.

> NTLMSSP Challenge

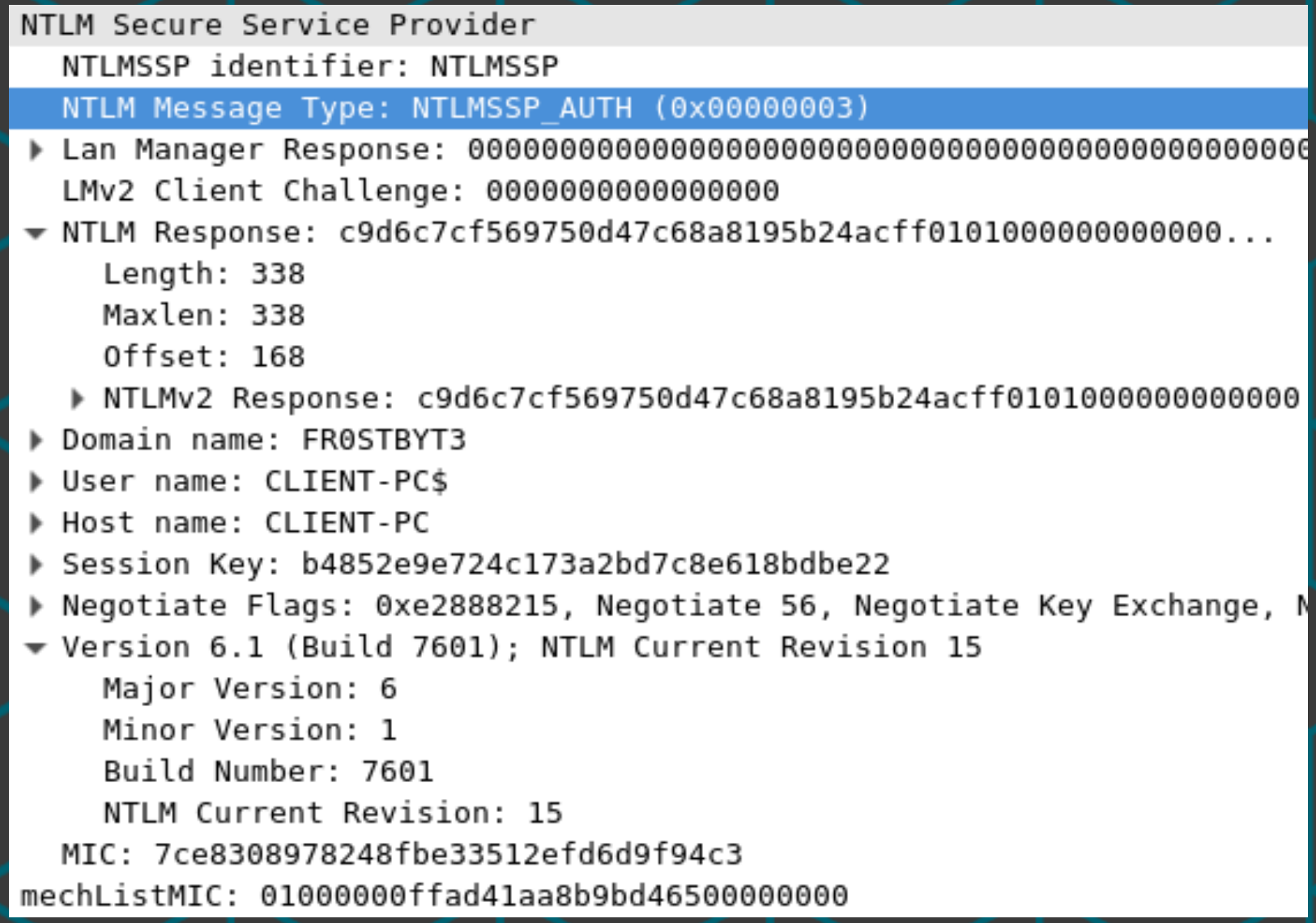


"Cool, hash your password & other data with this challenge"

Contains:

- Server challenge
- Server information

> NTLMSSP AUTH (Challenge-Response)



"Sure, here's everything"

Contains:

- Client username
- Client domain
- Client challenge
- NtProofString
- ResponseServerVersion
- HiResponseServerVersion
- A Timestamp (aTime)
- Encrypted new SessionBaseKey (Probably)

Great…so how is the SessionBaseKey generated?

> First – A Brief HMAC/CMAC Refresher

HMACs are keyed-hash message authentication code algorithms. Only those in possession of the private key, and the data, can generate the appropriate hash. These are used to verify the integrity of a message. CMACs are similar to HMACs with the exception that they're based on cipher block algorithms like AES rather than hashing algorithms like MD5.

Message = "Pineapple belongs on pizza"

SecretKey = "IwillDieOnThisHill"

HMAC_MD5(SecretKey, Message) = "34c5092a819022f7b98e51d3906ee7df"

> SessionBaseKey Generation

Let’s generate the SessionBaseKey from an NTLMv2 authentication process

Step #1:

NTResponse = HMAC_MD5(User’s NT Hash, username + domain)

> SessionBaseKey Generation

Step #2:

```
basicData       =       serverChallenge + responseServerVersion +
                                hiResponseServerVersion +  '\x00' * 6 +
                                aTime + clientChallenge + '\x00' * 4 +
                                serverInfo + '\x00' * 4


NtProofString           = HMAC_MD5(NTResponse, basicData)
```

> SessionBaseKey Generation

Step #3 (Last Step):

SessionBaseKey = HMAC_MD5(NTResponse, NtProofString)

> SessionBaseKey Generation

Put together:

NTResponse = HMAC_MD5(User's NT Hash, username + domain)

basicData = serverChallenge + responseServerVersion + hiResponseServerVersion + '\x00' * 6 + aTime + clientChallenge + '\x00' * 4 + serverInfo + '\x00' * 4

NtProofString = HMAC_MD5(NTResponse, basicData)

SessionBaseKey = HMAC_MD5(NTResponse, NtProofString)

So what information do we NOT have?

> SessionBaseKey Generation

Put together:

NTResponse = HMAC_MD5(**User's NT Hash**, username + domain)

basicData = serverChallenge + responseServerVersion + hiResponseServerVersion + '\x00' * 6 + aTime + clientChallenge + '\x00' * 4 + serverInfo + '\x00' * 4

NtProofString = HMAC_MD5(NTResponse, basicData)

SessionBaseKey = HMAC_MD5(NTResponse, NtProofString)

> SessionBaseKey Generation

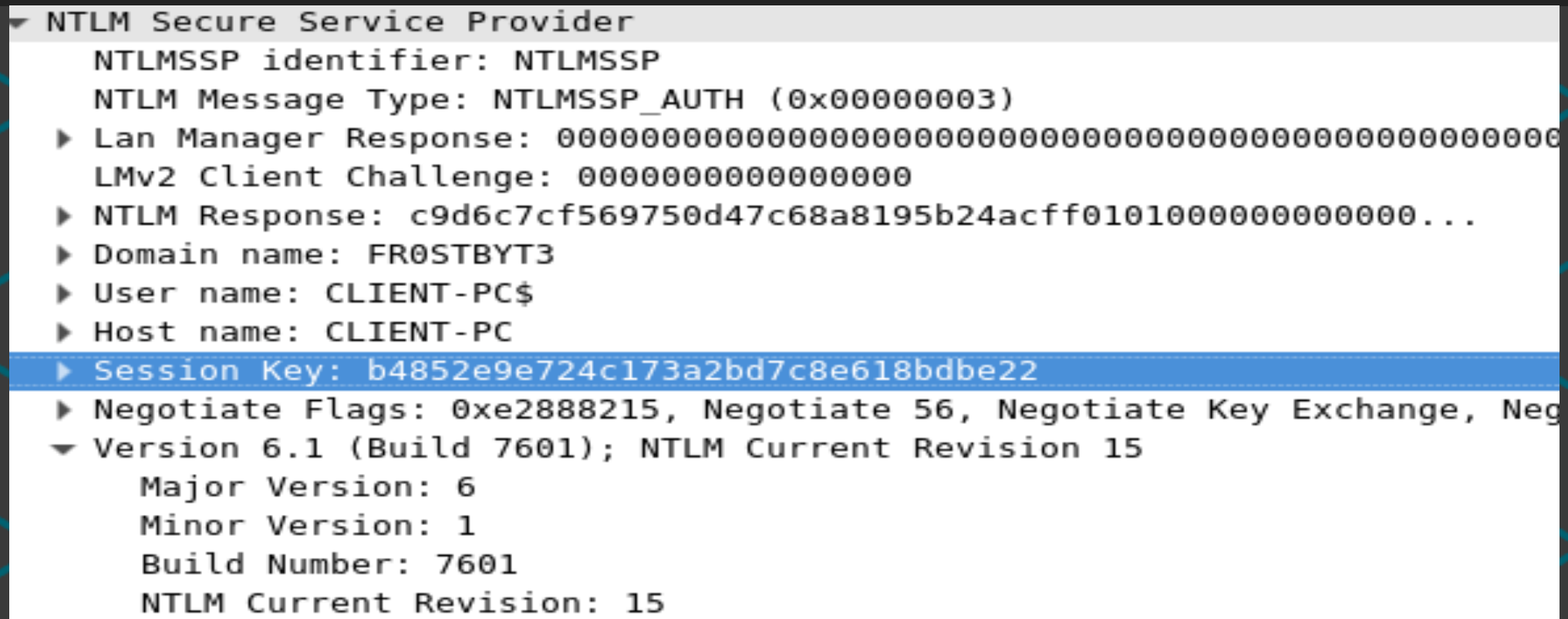
NTLM key logic:

User’s password -> Users’ NT Hash -> (Combined with challenge & auth data) -> SMB SessionBaseKey

If the user’s password is known, **SMB signatures can be forged**

> SessionBaseKey Generation

If “Negotiate Key Exchange” is set, then the client generates an entirely random new SessionBaseKey, RC4 encrypts it with the original SessionBaseKey, and sends it in the NTLMSS_AUTH request in the “SessionKey” field.



> SessionBaseKey Generation

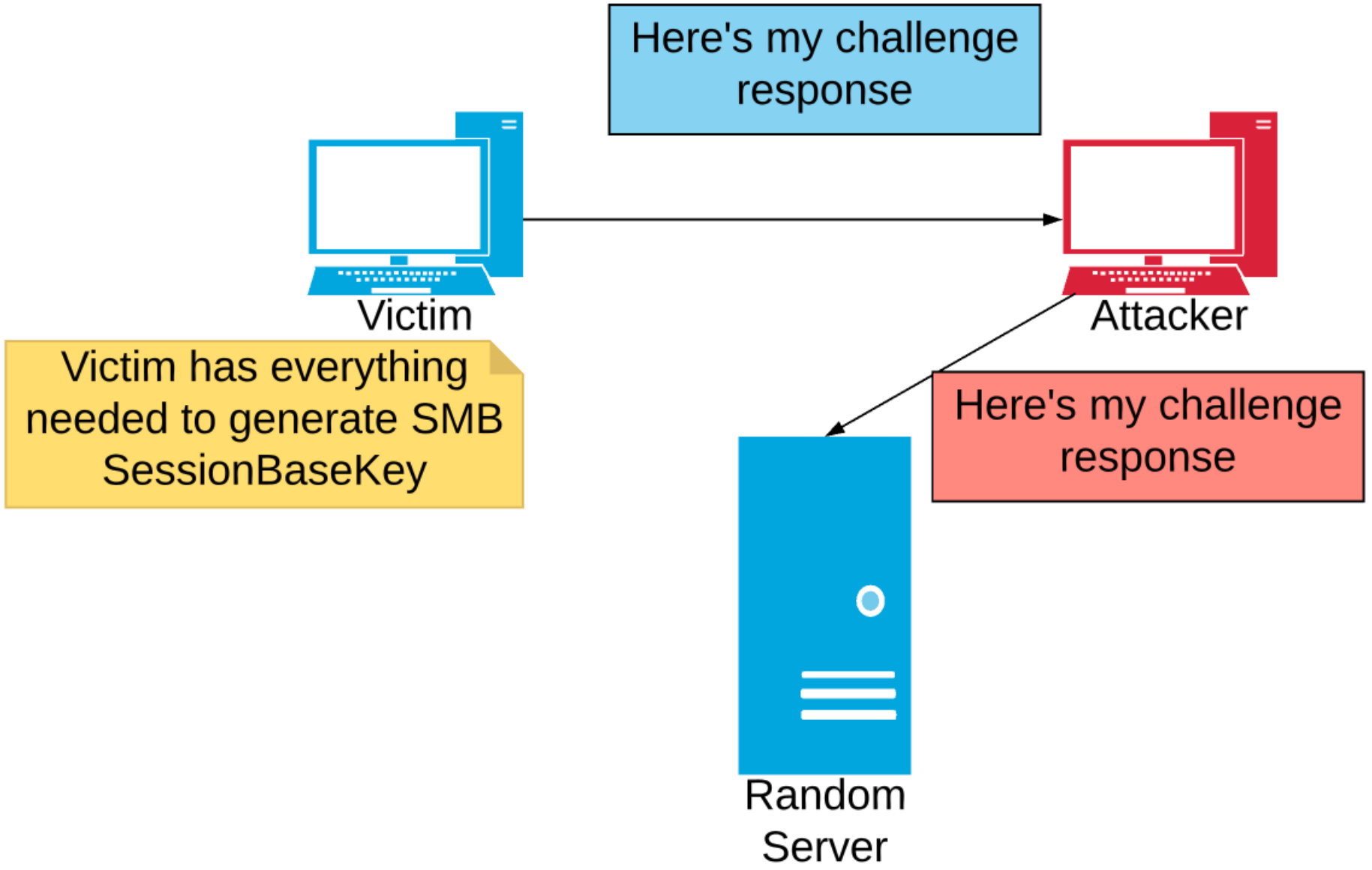
NTLM key logic with “Negotiate Key Exchange” :

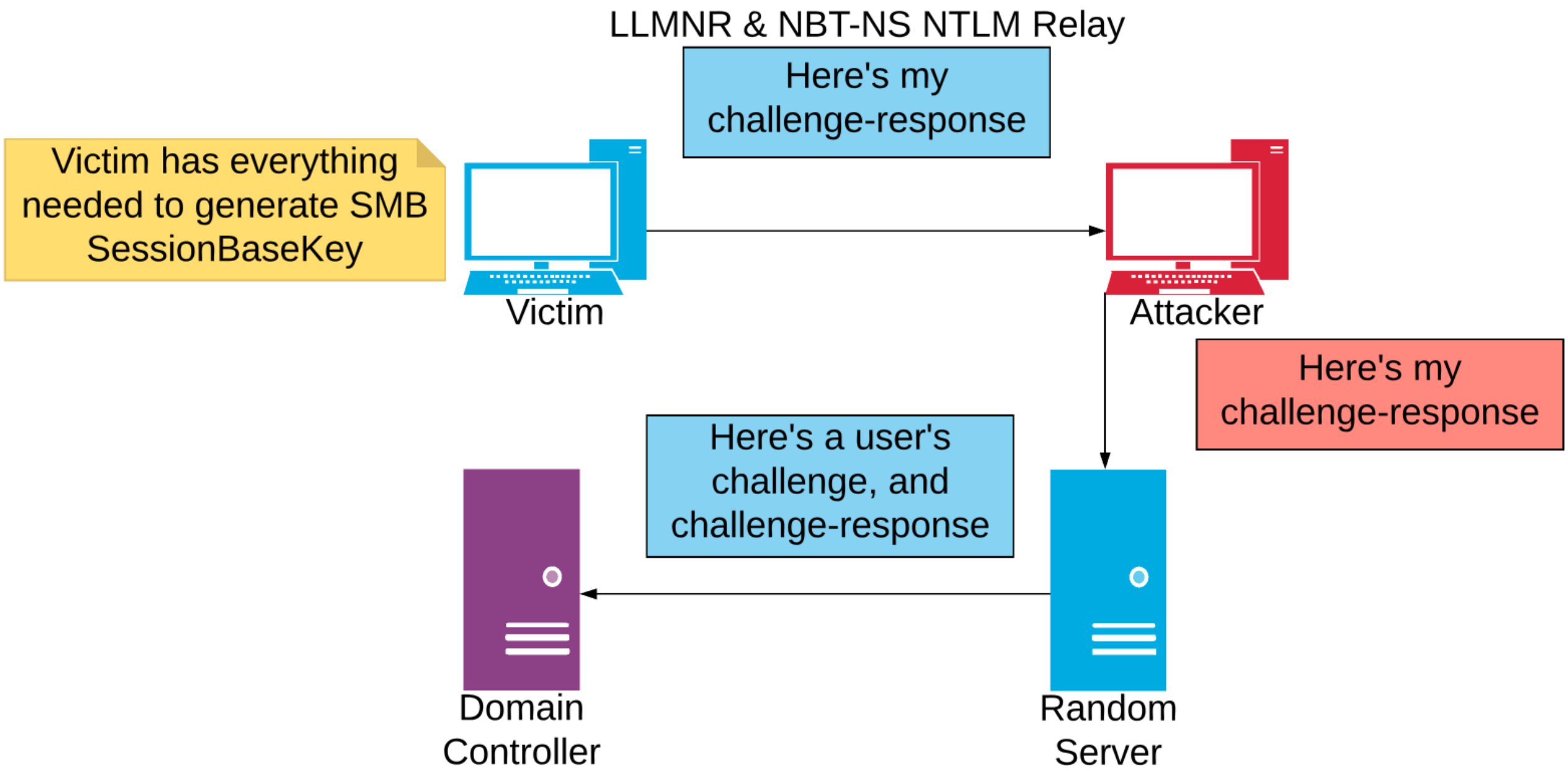
User’s password -> Users’ NT Hash -> (Combined with challenge & auth data) -> SMB SessionBaseKey -> SMB Exchanged SessionKey (becomes new SessionBaseKey)

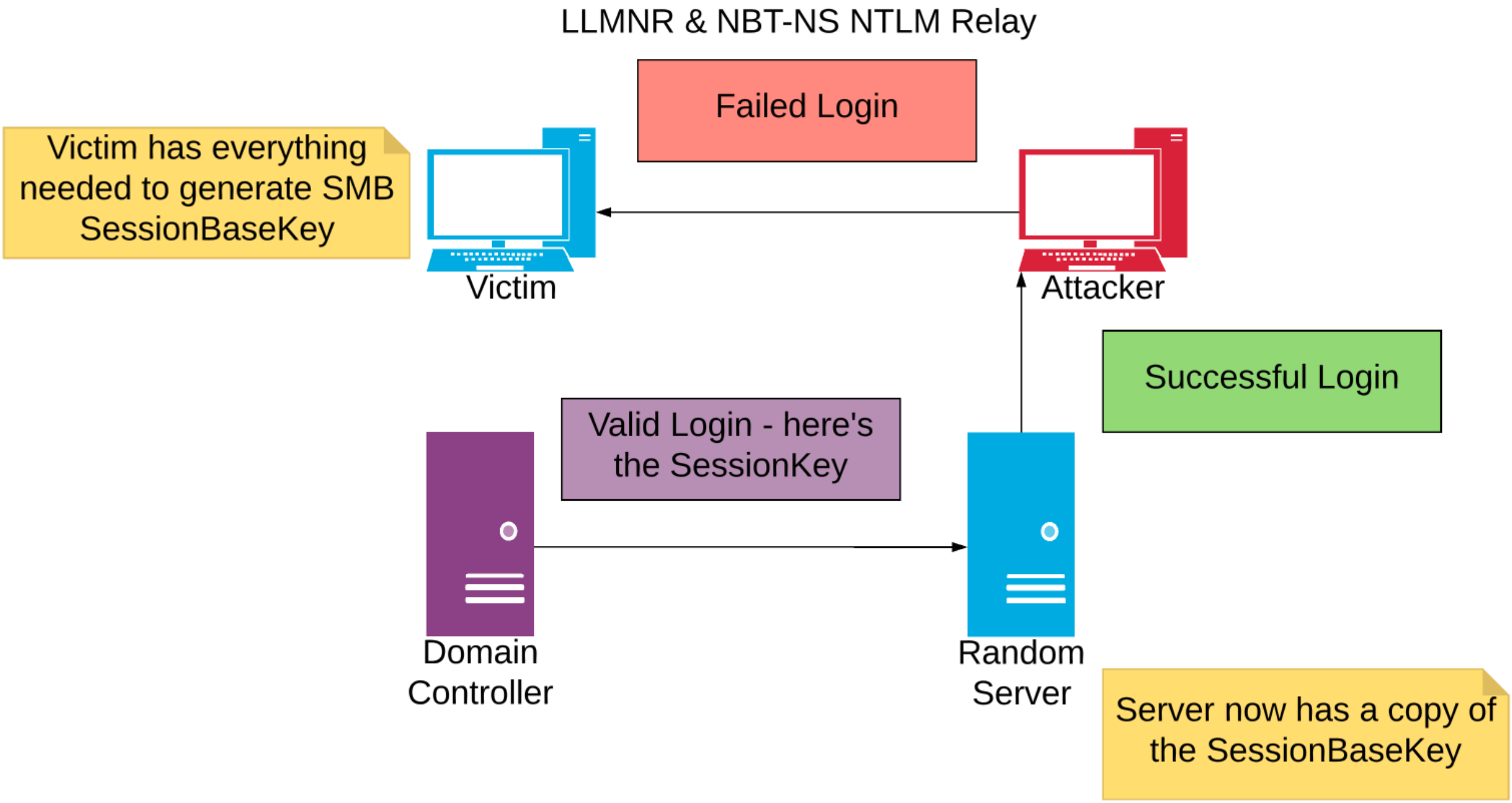
> The (now obvious) missing piece of the puzzle

To fill in the blanks from the NTLMv2 relay attack presented earlier

LLMNR & NBT-NS NTLM Relay




LLMNR & NBT-NS NTLM Relay
Here's my challenge-response
Victim has everything needed to generate SMB SessionBaseKey
Victim
Attacker
Here's my challenge-response
Here's a user's challenge, and challenge-response
Domain Controller
Random Server

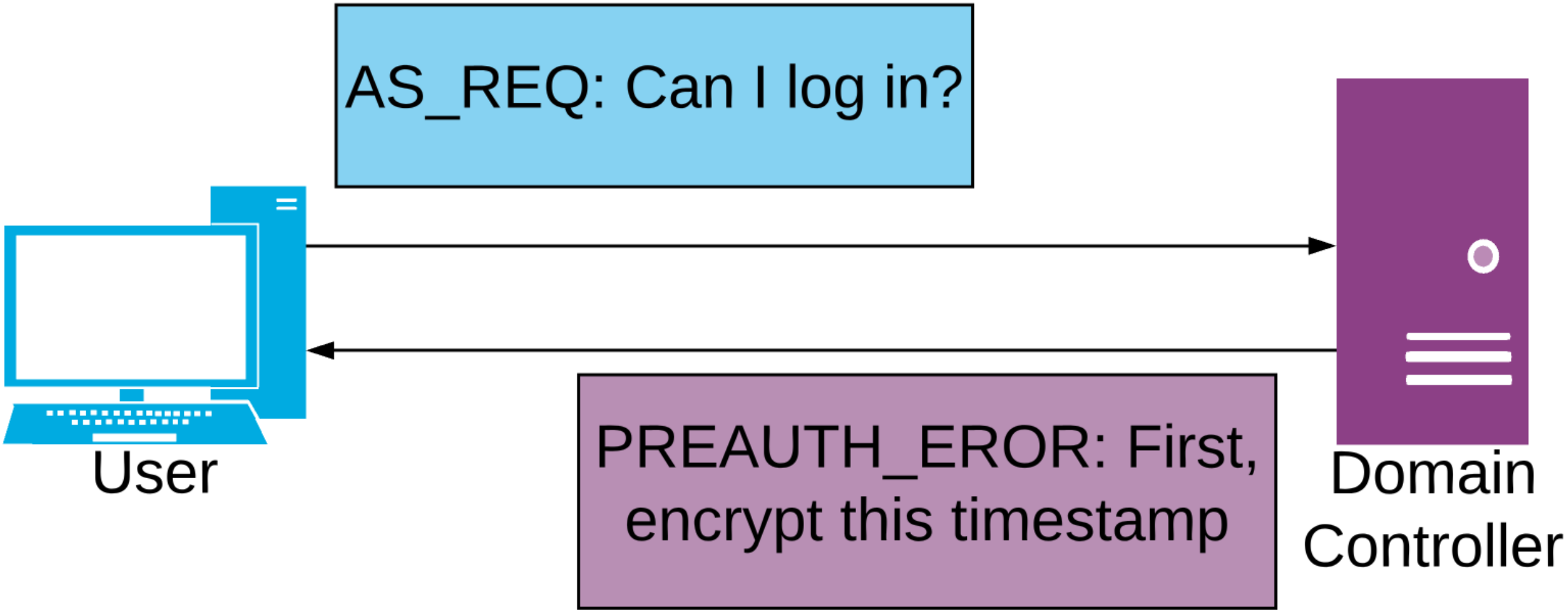

LLMNR & NBT-NS NTLM Relay
Failed Login
Victim has everything needed to generate SMB SessionBaseKey
Victim
Attacker
Successful Login
Valid Login - here's the SessionKey
Domain Controller
Random Server
Server now has a copy of the SessionBaseKey

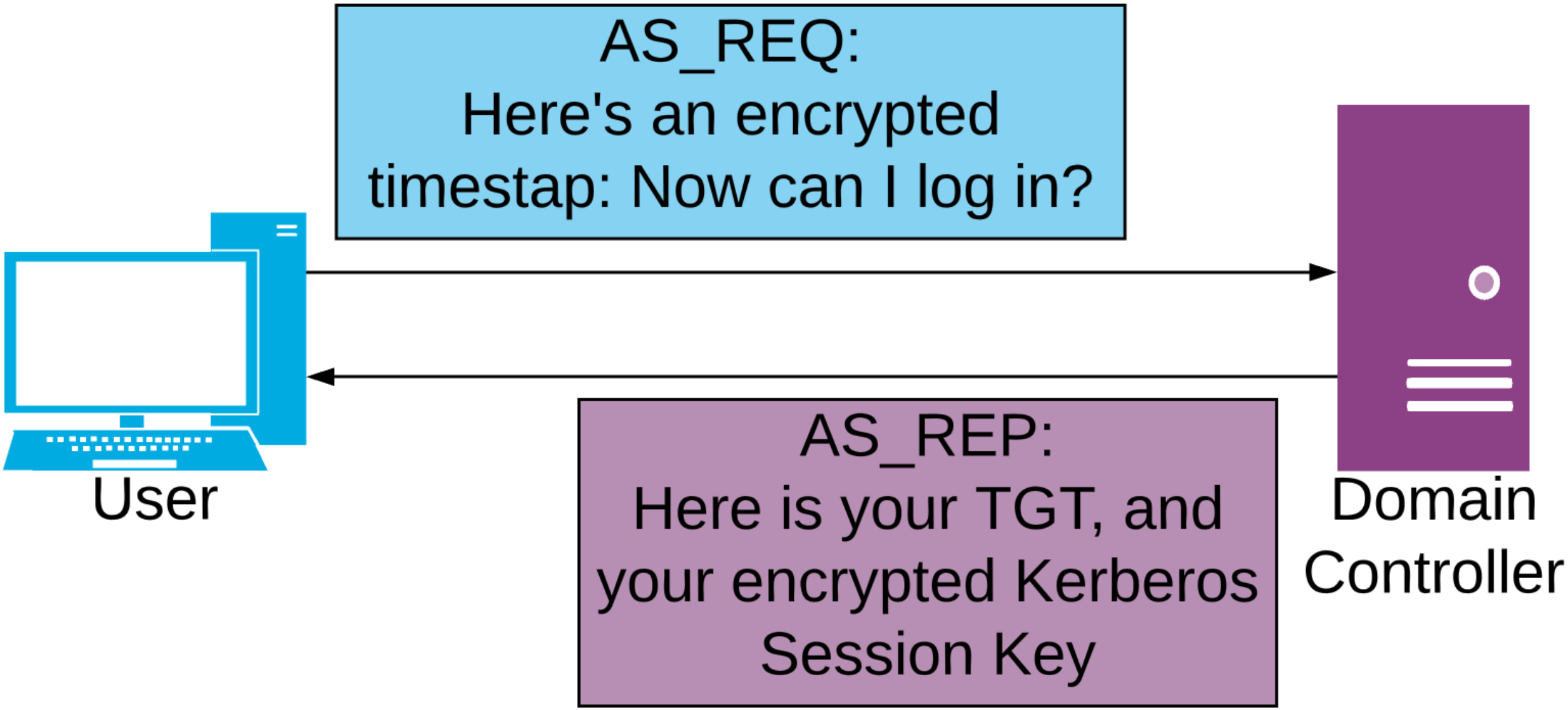
> The (now obvious) missing piece of the puzzle

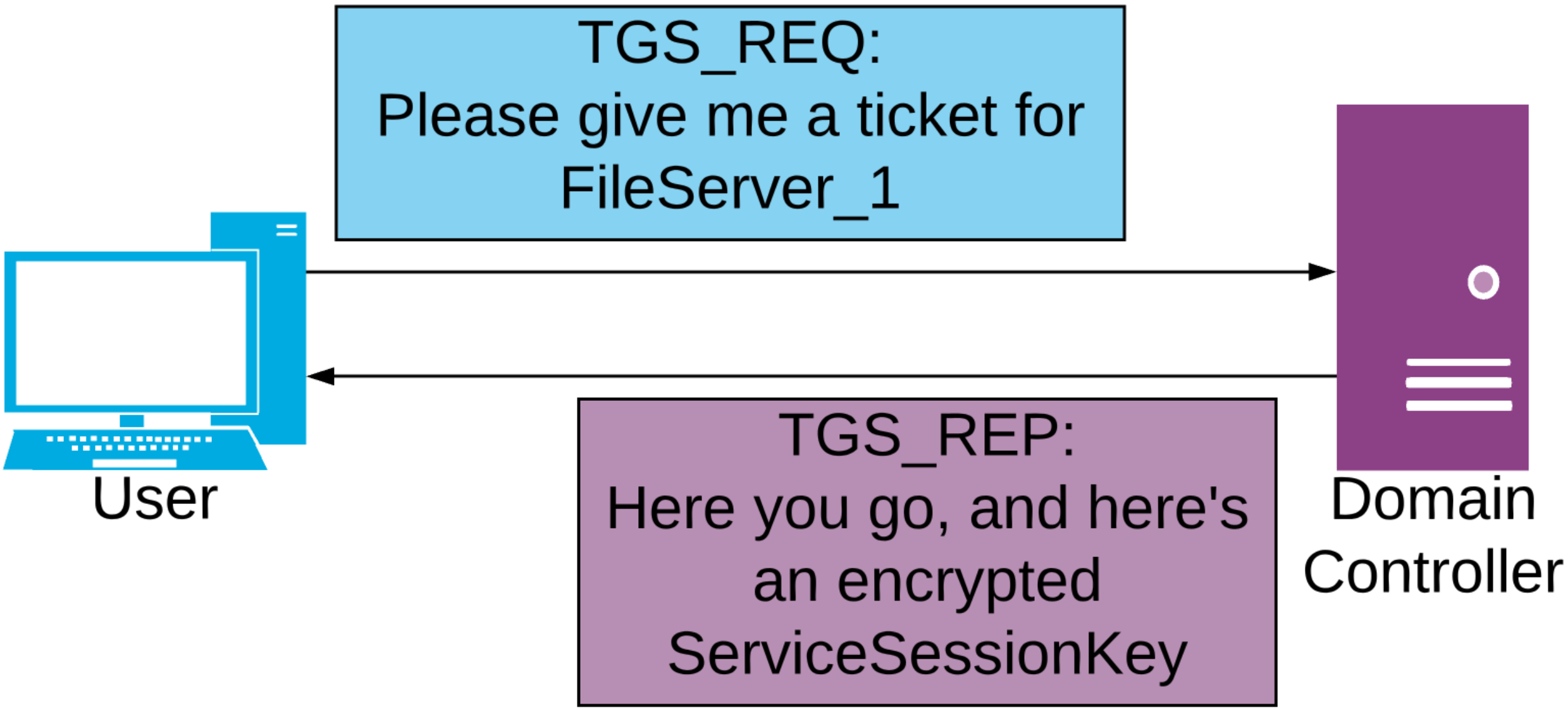
The DC takes in the challenge and the challenge response to generate the SessionBaseKey, and then sends it to the server through a required secure & encrypted channel.

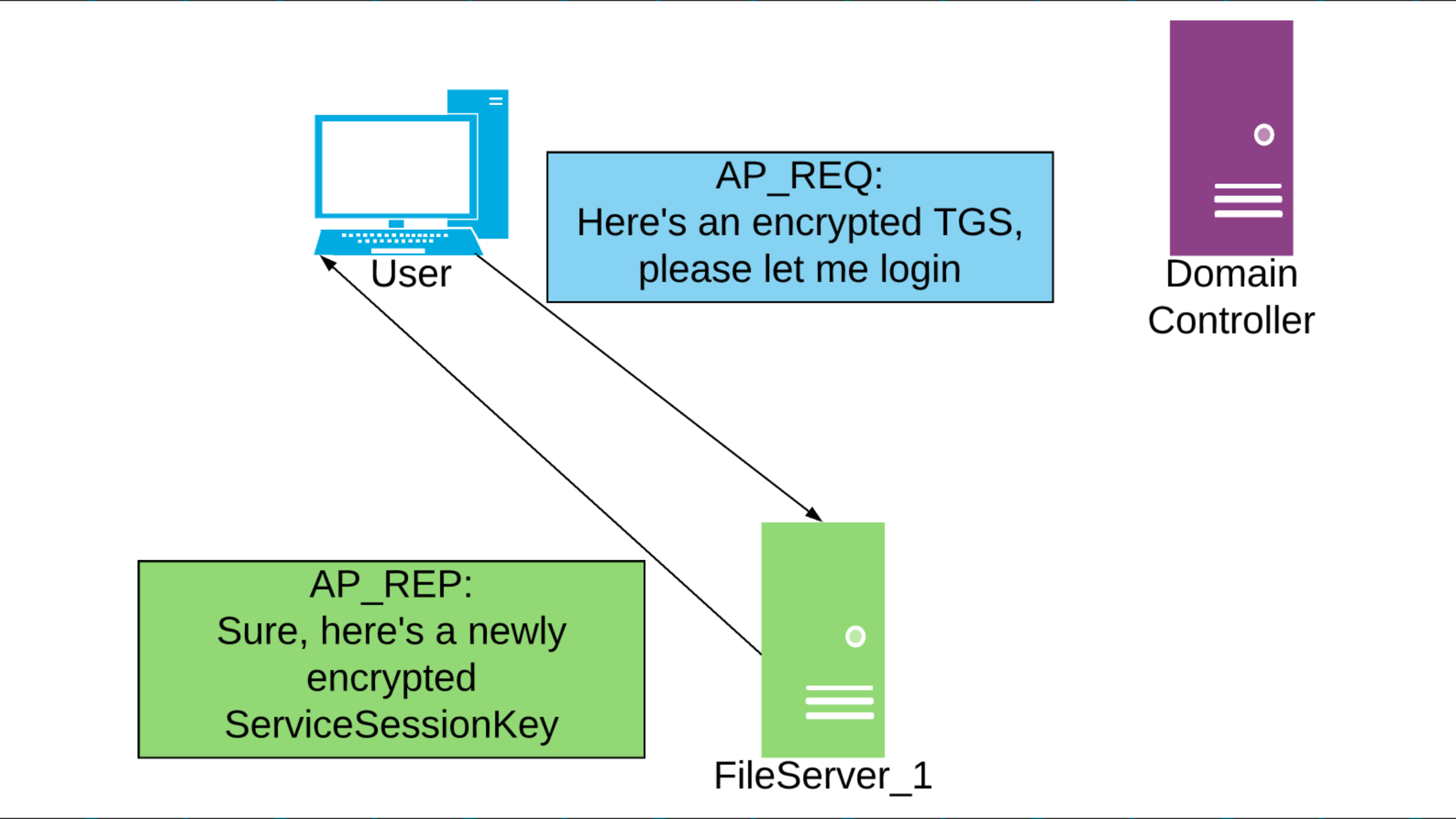
This secure channel is established by the NETLOGON service on a domain connected PC at startup, and the underlying authentication protecting it is the password for the machine account itself.

But wait, what about Kerberos?


AS_REQ: Can I log in?
User
PREAUTH_EROR: First, encrypt this timestamp
Domain Controller


AS_REQ:
Here's an encrypted timestap: Now can I log in?
User
Domain Controller
AS_REP:
Here is your TGT, and your encrypted Kerberos Session Key


TGS_REQ:
Please give me a ticket for FileServer_1
User
Domain Controller
TGS_REP:
Here you go, and here's an encrypted ServiceSessionKey


User
AP_REQ:
Here's an encrypted TGS,
please let me login
Domain
Controller
AP_REP:
Sure, here's a newly
encrypted
ServiceSessionKey
FileServer_1

> What about Kerberos?

The SessionBaseKey is the ServiceSessionKey in the TGS response.

Key logic:

User’s plaintext password -> Kerberos Session Key -> Service Session Key (inside TGS)

> What about Kerberos?

If mutual authentication is used, the server will reply with a new SessionBaseKey which is encrypted with the previous one.

Key logic:

User’s plaintext password -> Kerberos Session Key -> Service Session Key (inside TGS) -> New Service Session Key (inside AP_REP)

> Signing the packet

Once we have the SessionBaseKey, we can sign the packet

SMB 1.0

Signature = HMAC_MD5(SessionBaseKey, SMBPacket)

SMB 2.0.2 & 2.1.0

Only the first 16 bytes of the hash make up the signature

Signature = HMAC_SHA256(SessionBaseKey, SMBPacket)

SMB 3.0.0, 3.0.2 & 3.1.1

A special signing key is derived from the SessionBaseKey

Signature = CMAC_AES128(SigningKey, SMBPacket)

> Attacking SMB

> Attacking SMB

So, what happens when we know the SessionBaseKey, or when signing is not required at all **per the default settings**?

HTTP before HTTPS

> Attacking SMB

- If encryption is not used
  - Steal copies of files passed over the wire in cleartext

- If signing is not used
  - Replace every file with an identical LNK that executes our code
  - Swap out the contents of any legitimate file with our own malicious one of the same file-type
  - Inject fake files into directory listings (for social engineering)

- If signing is used, and the attacker knows the key
  - All above + backdoor any logon scripts & SYSVOL data we can

> Attacking SMB

(Defaults) SMB1:	Client supports it, server doesn't. Unless they both support it, or one requires it, **no signing will be used**.
(Defaults) SMB2/3:	Server & client support it, but don't require it. **Unless someone requires it, signing is not used**.

> Attacking SMB

(Defaults) SMB1:	No encryption support
(Defaults) SMB2/3:	Encryption was introduced in SMB3 but must be manually enabled or required

Non-DC Machines	SMB1 Client Signing	SMB1 Server Signing	SMB2/3 Client Signing	SMB2/3 Server Signing	Highest Version
Windows Vista SP1	Supported, Not required	Disabled	Not required	Not required	2.0.2
Windows 7	Supported, Not required	Disabled	Not required	Not required	2.1.0
Windows 8	Supported, Not required	Disabled	Not required	Not required	3.0.0
Windows 8.1	Supported, Not required	Disabled	Not required	Not required	3.0.2
Windows 10*	Supported, Not required	Disabled	Not required	Not required	3.1.1
Server 2008	Supported, Not required	Disabled	Not required	Not required	2.0.2
Server 2008 R2	Supported, Not required	Disabled	Not required	Not required	2.1.0
Server 2012	Supported, Not required	Disabled	Not required	Not required	3.0.0
Server 2012 R2	Supported, Not required	Disabled	Not required	Not required	3.0.2
Server 2016*	Supported, Not required	Disabled	Not required	Not required	3.1.1

> Notable Exceptions

- Domain controllers require SMB signing by default.

- In Windows 10 and Server 2016, signing and mutual authentication (ie. Kerberos auth) is **always** required on *\SYSVOL and *\NETLOGON

  These default settings are enforced through UNC hardening

> Notable Exceptions

- If a client supports SMB3, and the dialect picked for the connection is SMB 2.0.2 -> 3.0.2, the client will **always** send a signed FSCTL_VALIDATE_NEGOTIATE_INFO message to validate the negotiated “Capabilities, Guid, SecurityMode, and Dialects” from the original negotiation phase – and require a signed response. This feature cannot be disable in Windows 10/Server 2016 and later.

  This prevents dialect downgrades (except to SMBv1), and stripping any signature/encryption support.

> Notable Exceptions

SMB 3.1.1 is a security **beast**. It leverages pre-authentication integrity hashing to verify that **all** information prior to authentication was not modified.

In a nutshell: It takes the cumulative SHA-512 hash of the every SMB packet prior to authentication, and uses it in the SessionBaseKey generation process. If any data was modified, then the client and server will not generate the same signing keys.

At the end of the session setup response, both client and server must send a signed message to prove integrity.


NOT BAD
quickmeme.com

> Recap

- Signing is not used by default except on DCs, and when W10/Server2016 connect to *\NETLOGON and *\SYSVOL
- Encryption is only supported in SMB 3.0.0 and greater, and must be enabled or required manually.
- Every dialect up except for SMB 3.1.1 can be downgraded to NTLMv2 if it is supported.
- Signing and encryption keys are, at their root, based on knowledge of the user's password.

Introducing SMBetray

> Attacking SMB

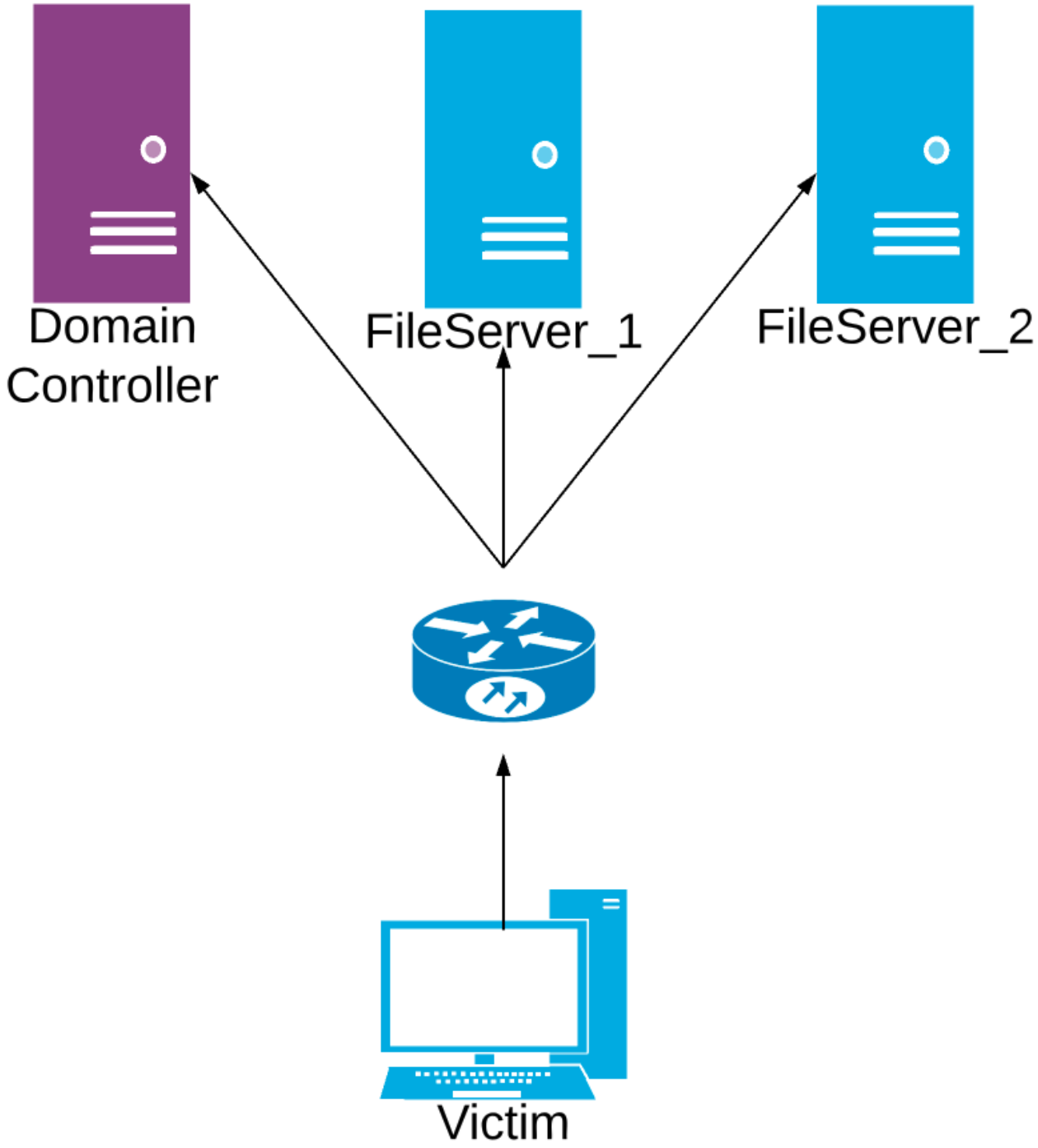
The goal was to build a tool to take full advantage of the gaps in security of weak SMB sessions from a true MiTM perspective.

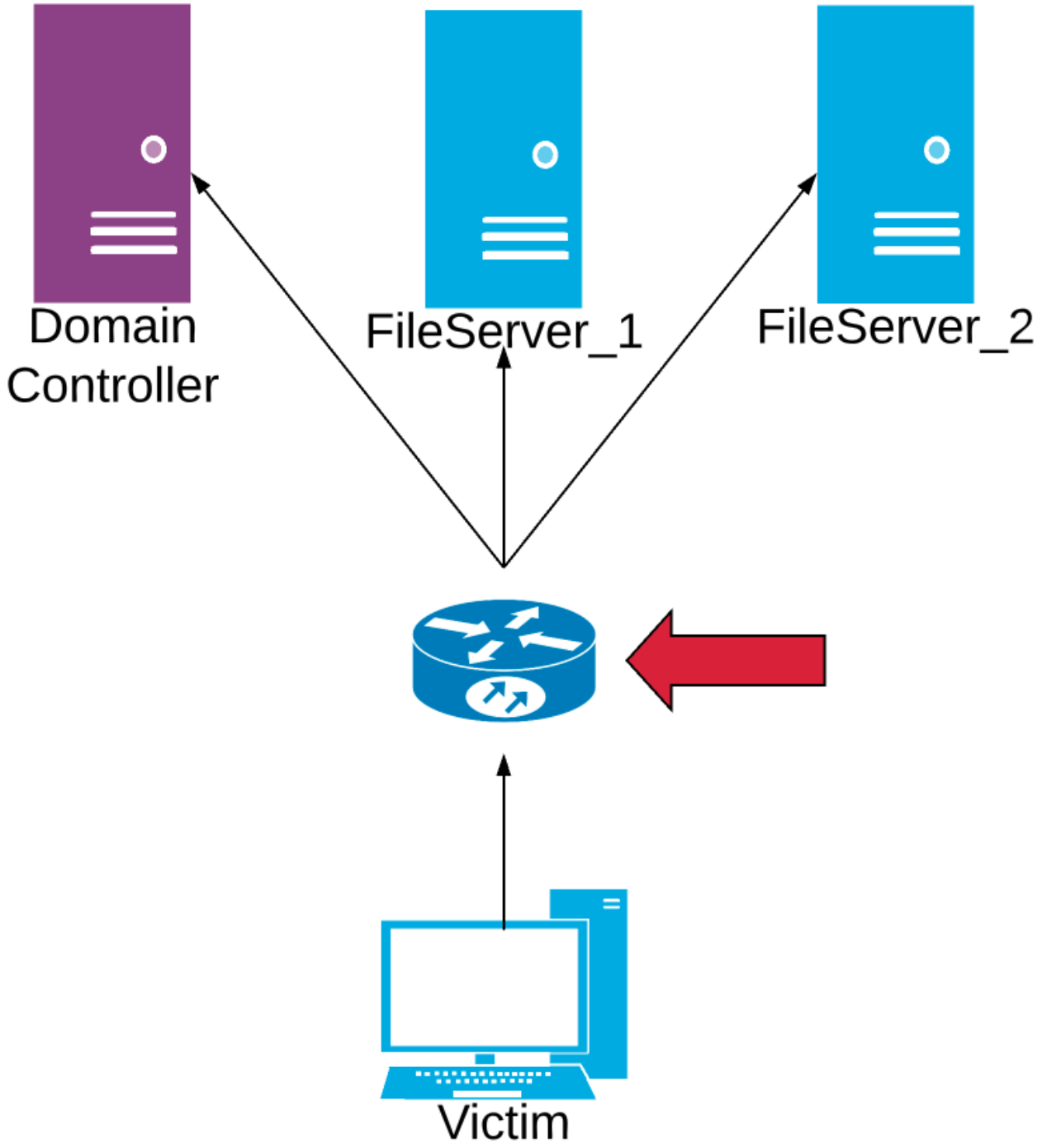
Primary objectives were to steal sensitive data, and gain RCE

> SMBetray

Biggest obstacle:

Putting the tool in the ideal position for intercept. The ideal position was a fully transparent intercept/proxy that, when a victim routes all traffic through us, would transparently eavesdrop on the connection between the victim and their actual legitimate destination.


Domain
Controller
FileServer_1
FileServer_2
Victim


Domain
Controller
FileServer_1
FileServer_2
Victim

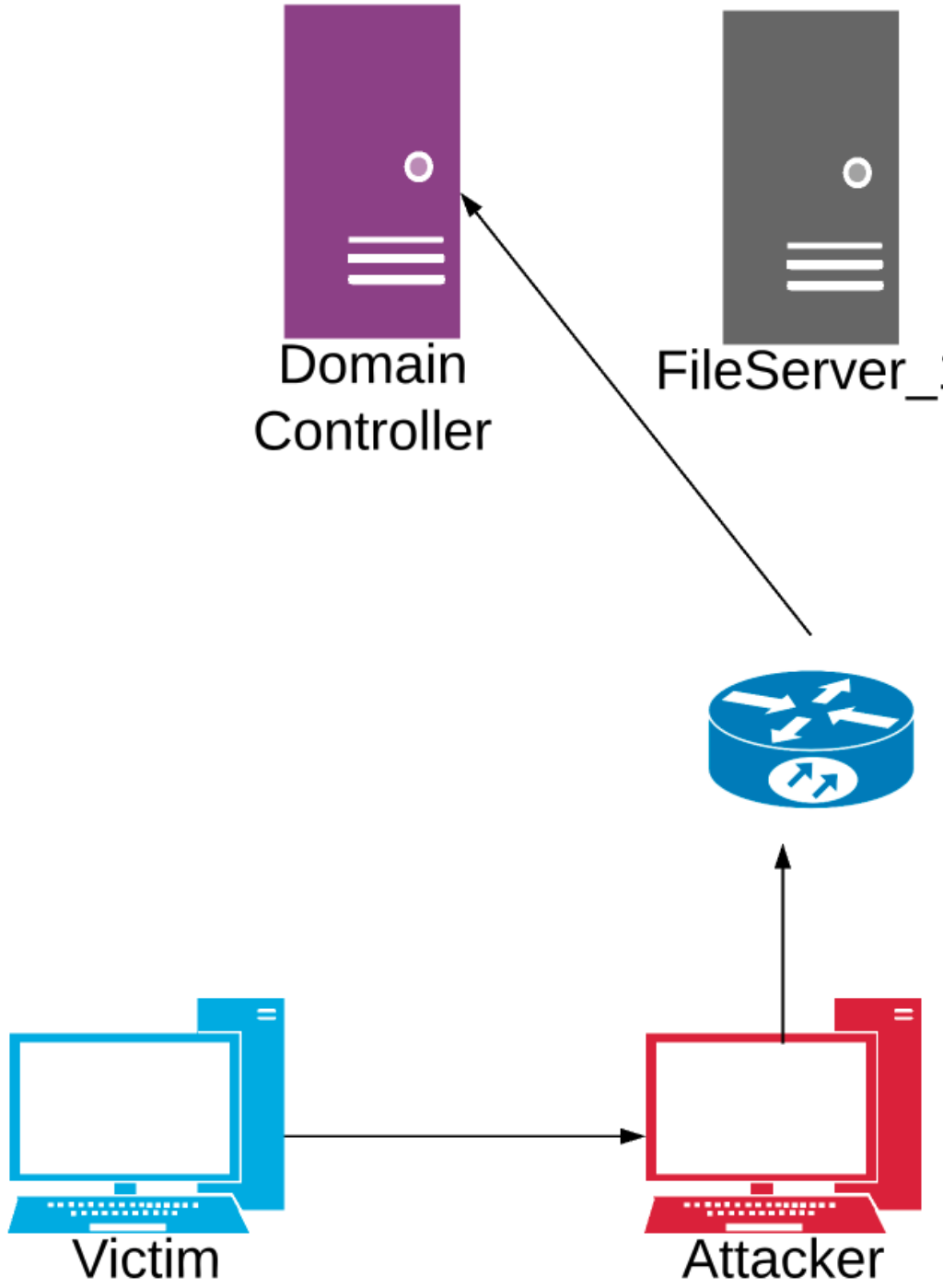
> SMBetray

Existing options provided two possibilities:*

1. Use of an arbitrary upstream server

2. Use of NFQUEUE to edit the packets at the kernel level

**Note: I did not flip the Internet upside down searching for the perfect solution*



Use of pre-configured upstream server

> SMBetray

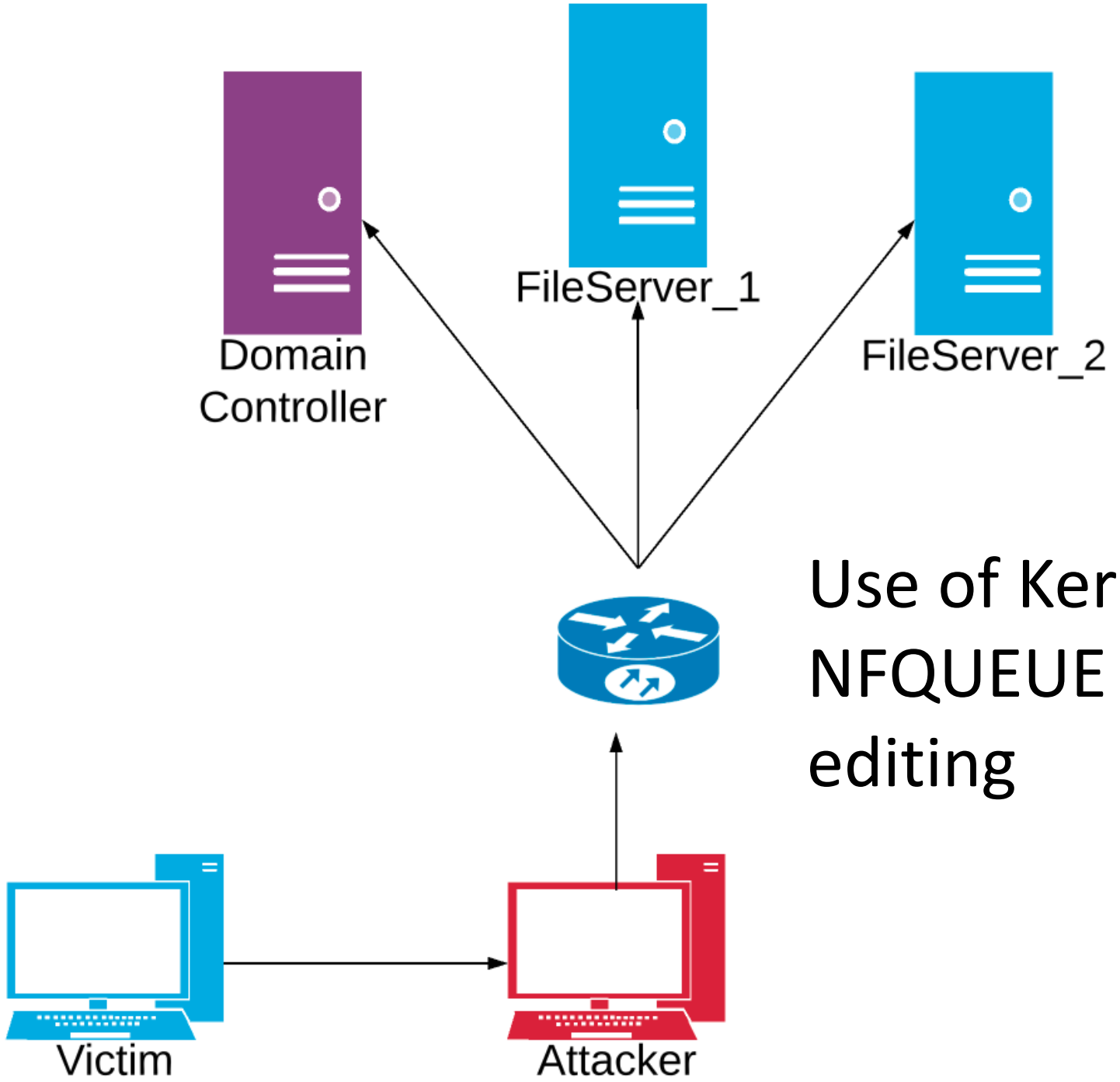
Use of a pre-configured upstream server

Pros:

- Connection stability

Cons:

- You're most likely redirecting the requests of your victim somewhere other than where they meant to go. This causes disruptions, and doesn't provide the true "transparent" MiTM setup desired. Once the data is re-directed through iptables, we lose the original "destination" information, so we can't determine where the victim was originally sending the request. (HTTP MiTM servers avoid this issue by grabbing the "Host:" info from the header)



Use of Kernel level NFQUEUE packet editing

> SMBetray

Use of a NFQUEUE

Pros:

- Full transparency, as packets aren't redirected – they're edited on the fly at the kernel level

Cons:

- This level of TCP packet editing takes too long (from a TCP timeout perspective). This quickly leads to TCP re-transmission snowballing, and connections resetting.

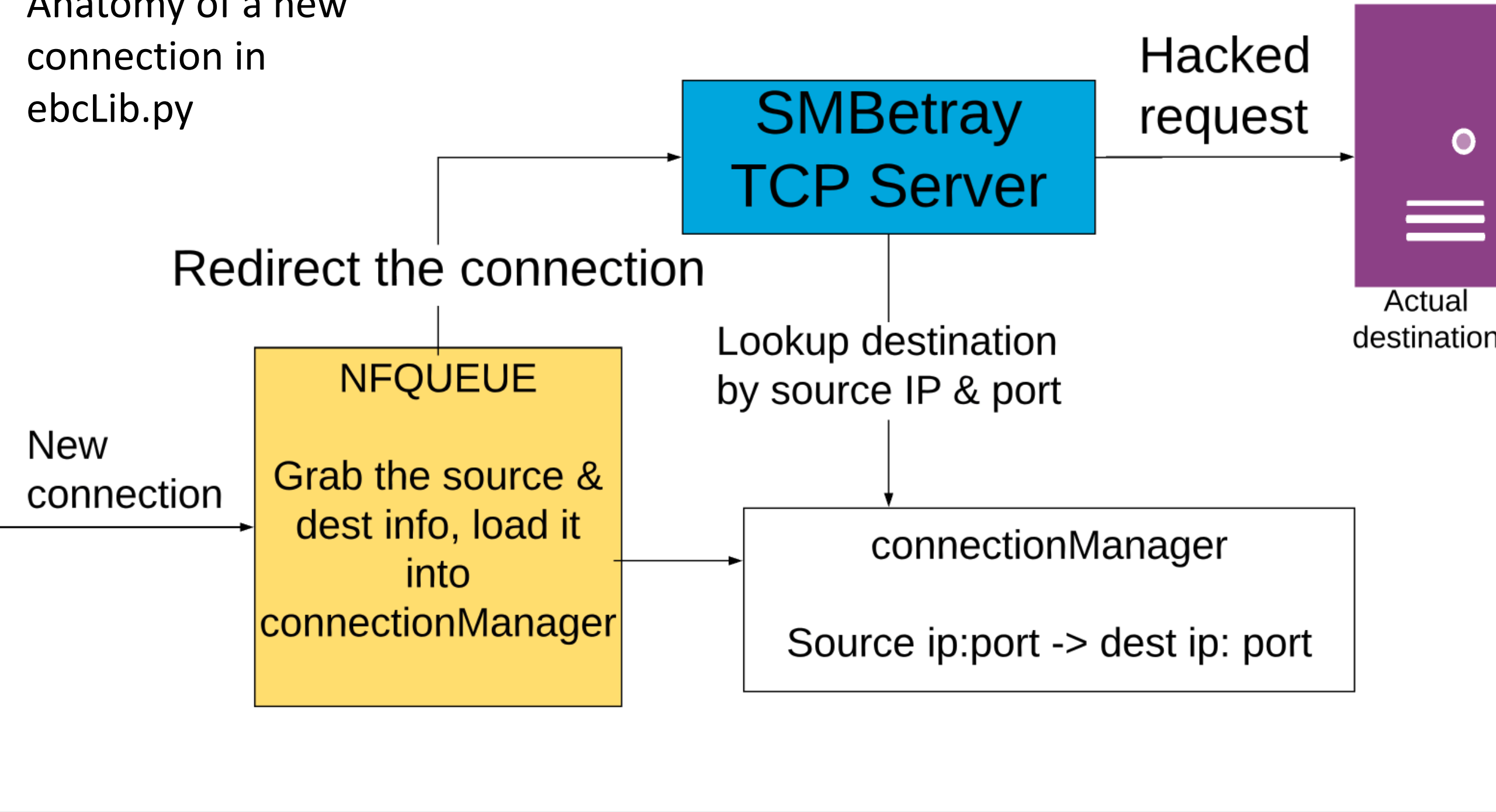
Solution? Combine them

> SMBetray

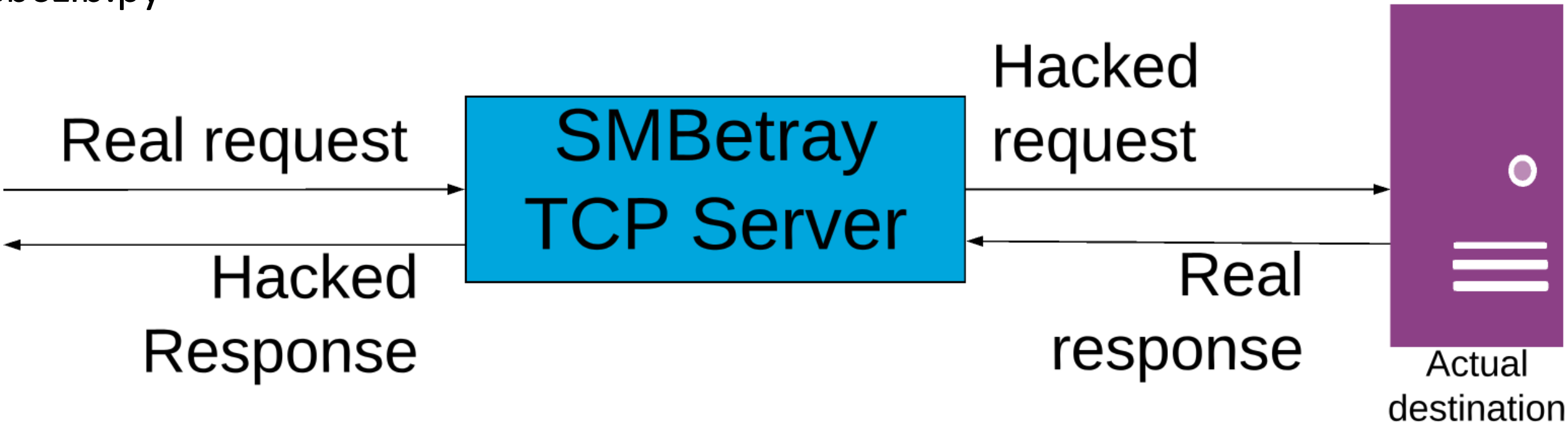
Created ebcLib.py as a MiTM TCP proxy with the useful transparency of an NFQUEUE intercept, with the connection stability of an upstream MiTM proxy.

SMBetray is built on top of this library.

Anatomy of a new connection in ebcLib.py



Anatomy of an existing/established connection in ebcLib.py

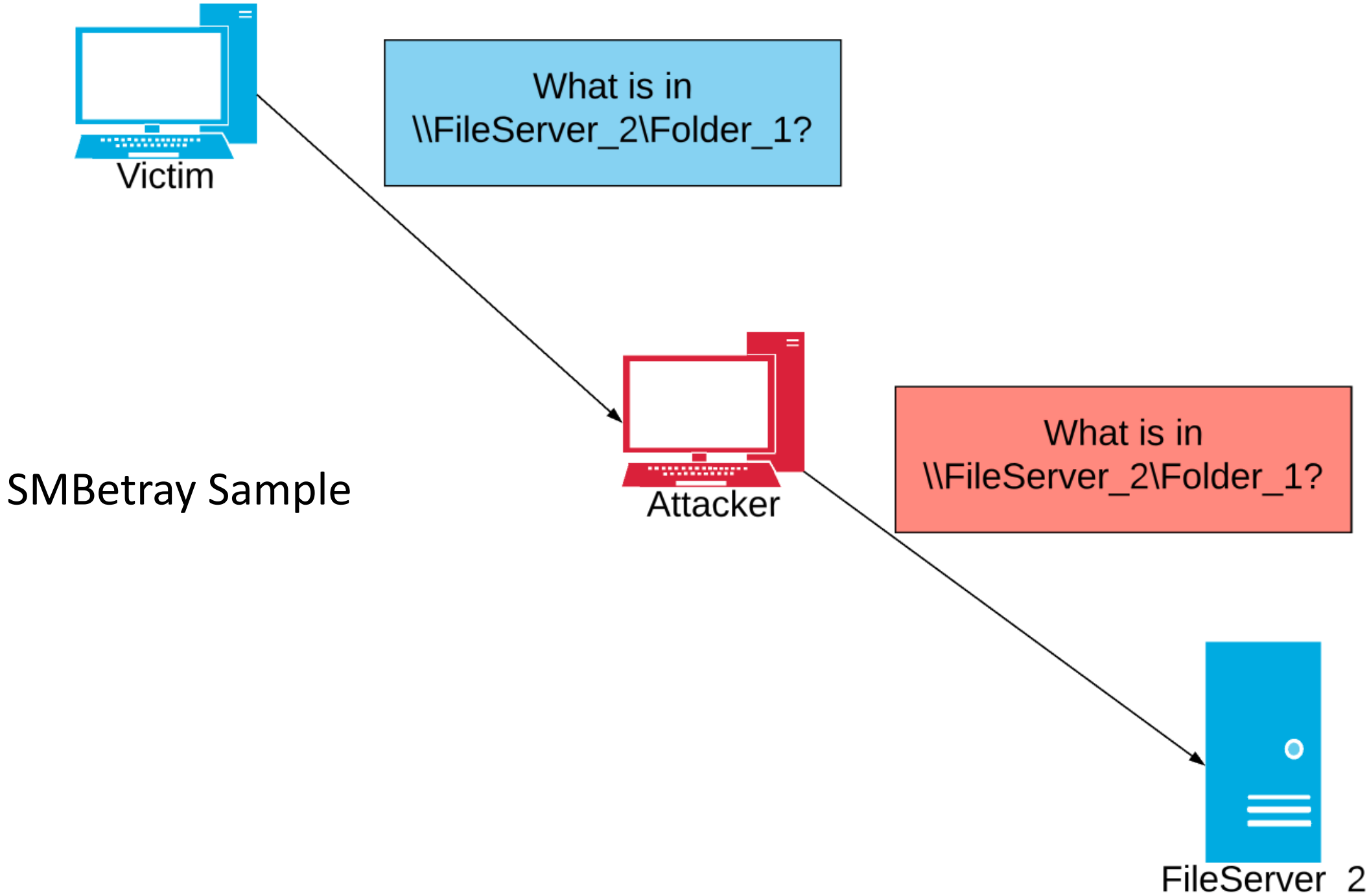


> SMBetray

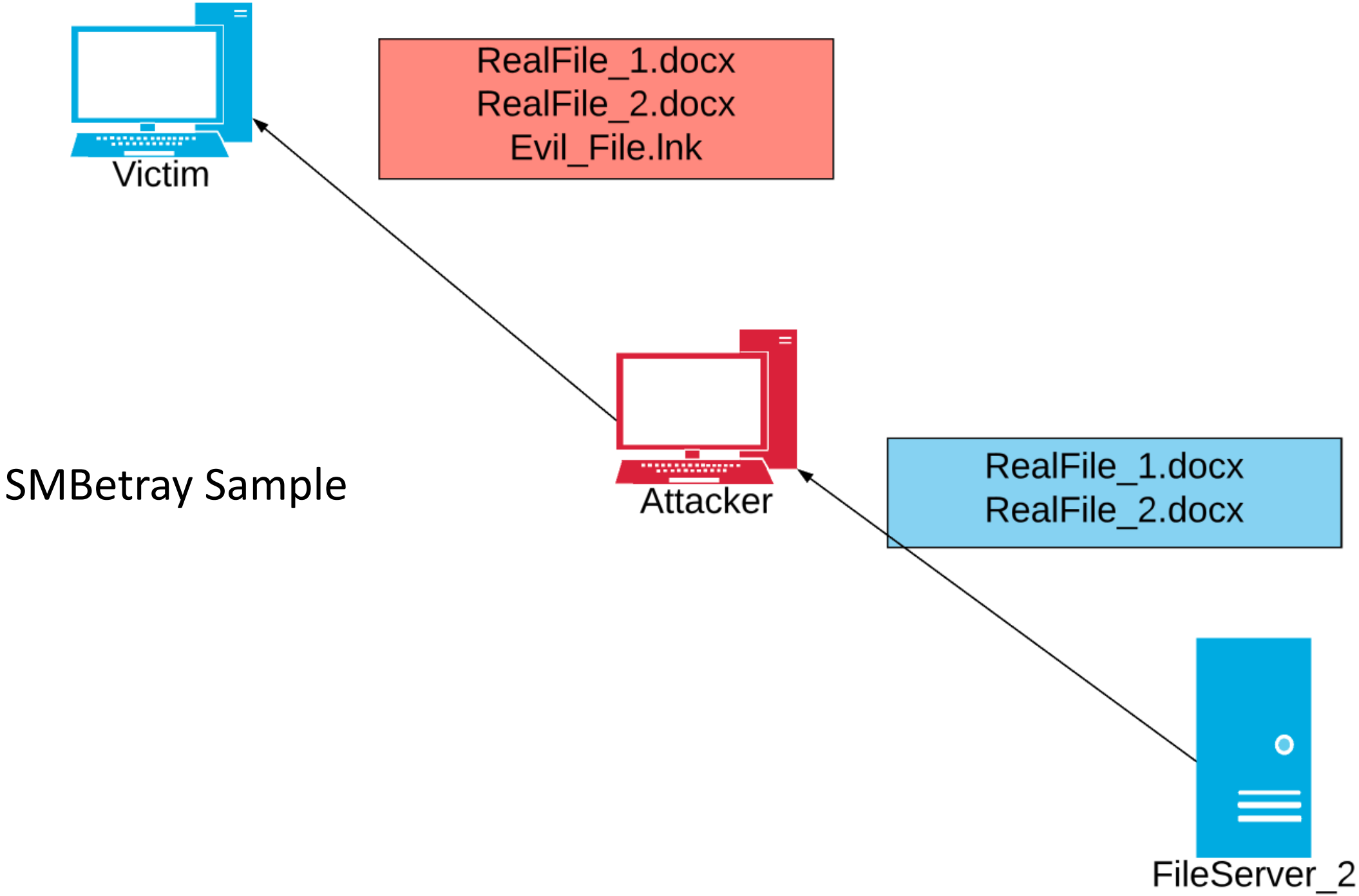
Now that we can put ourselves where we want, lets start attacking SMB

SMBetray runs the below attacks:

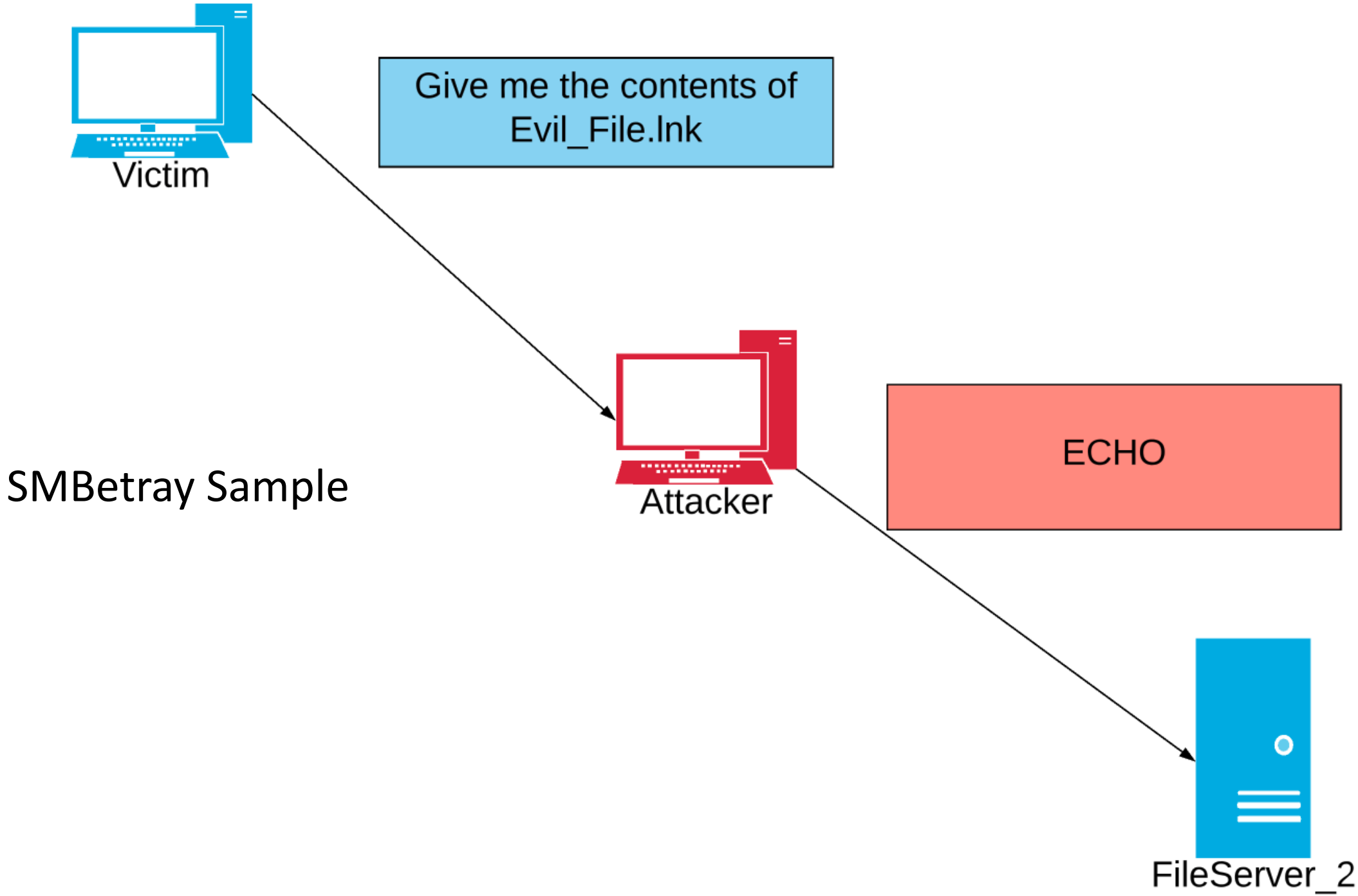
- Dialect downgrading & Authentication mechanism downgrading
- Passive file copying
- File directory injection
- File content swapping/backdooring
- Lnk Swapping
- Compromise SessionKeys (if we have creds) & forge signatures



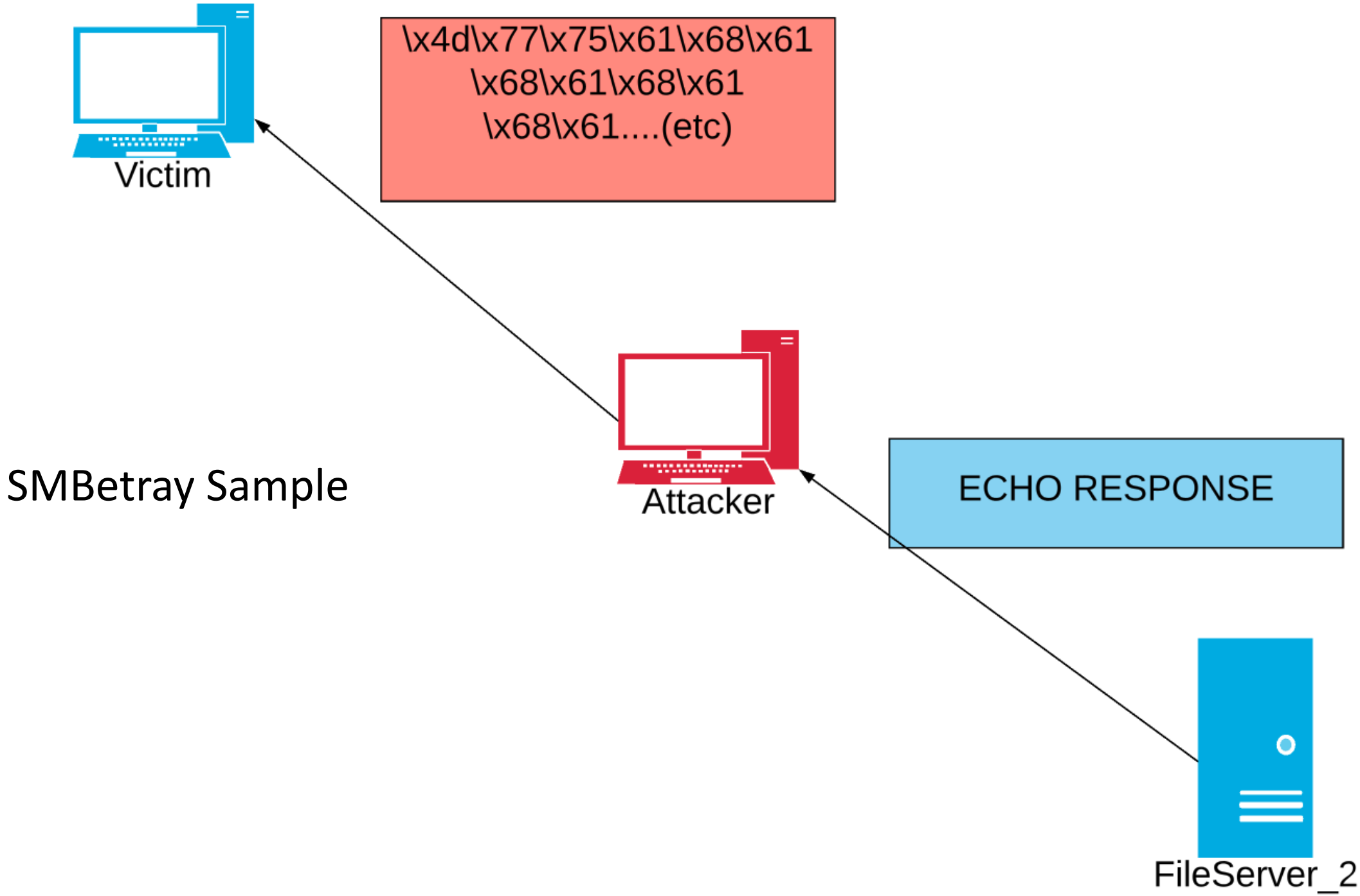
SMBetray Sample



SMBetray Sample

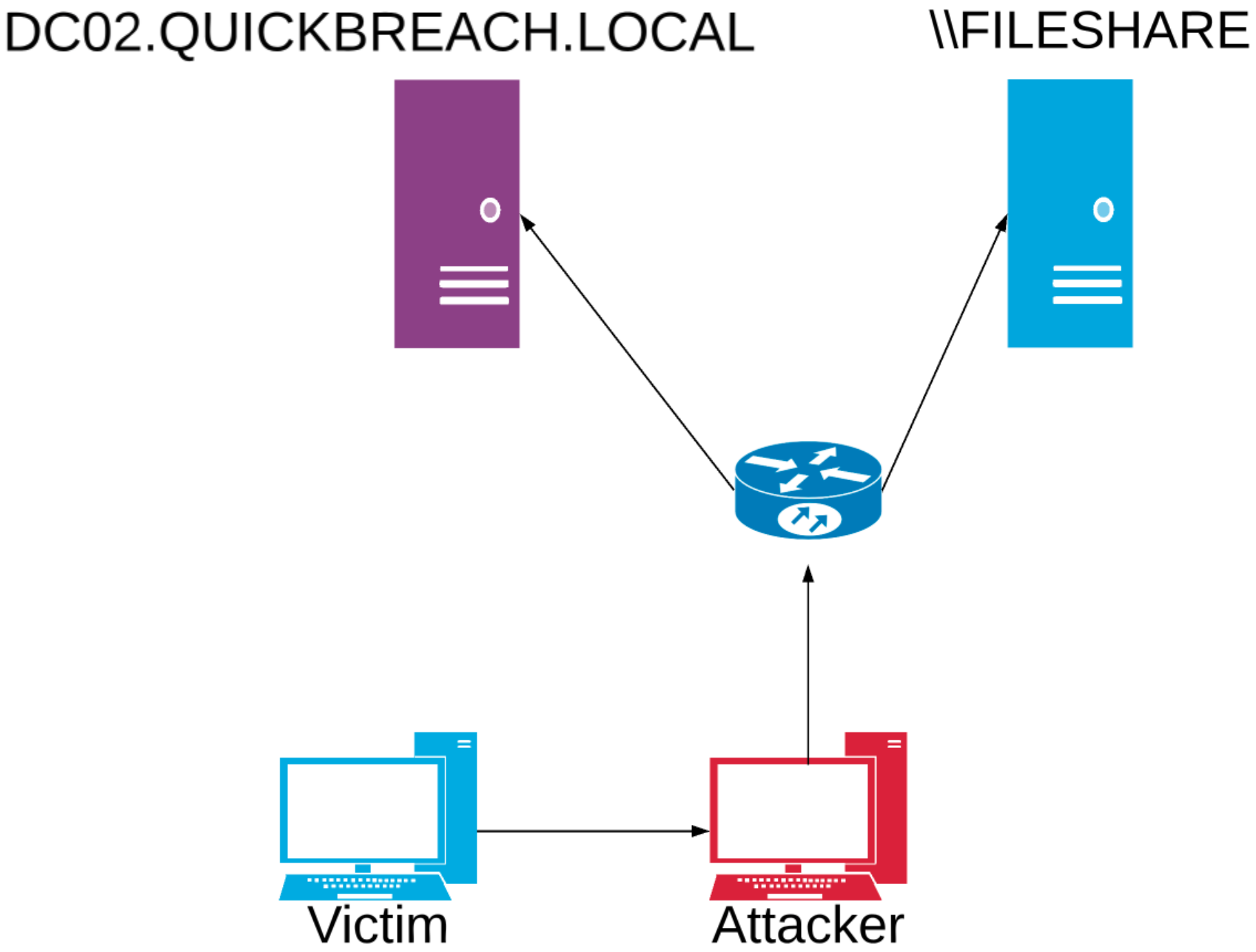


SMBetray Sample



SMBetray Sample

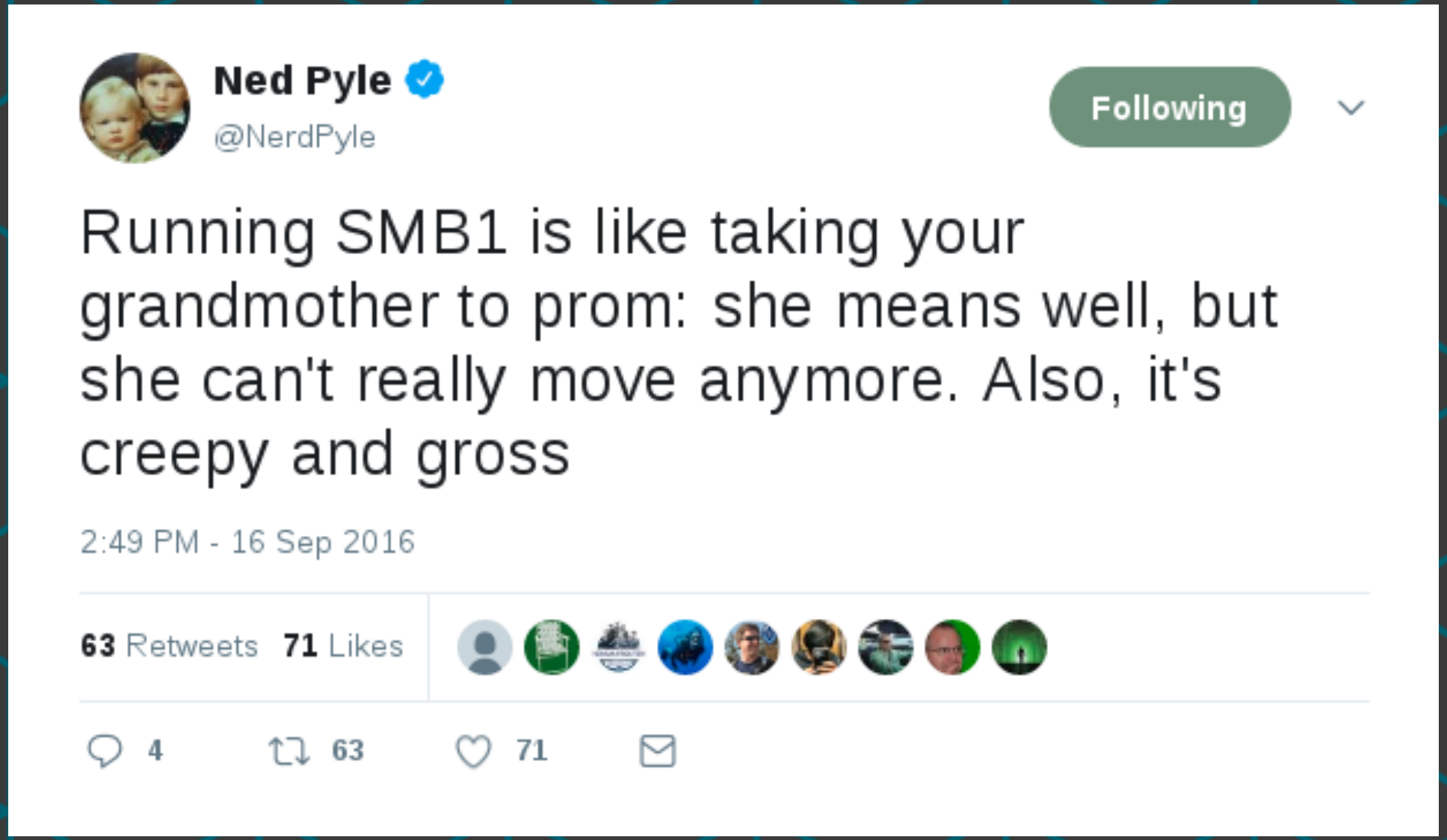
Demo


DC02.QUICKBREACH.LOCAL
\\FILESHARE
Victim
Attacker

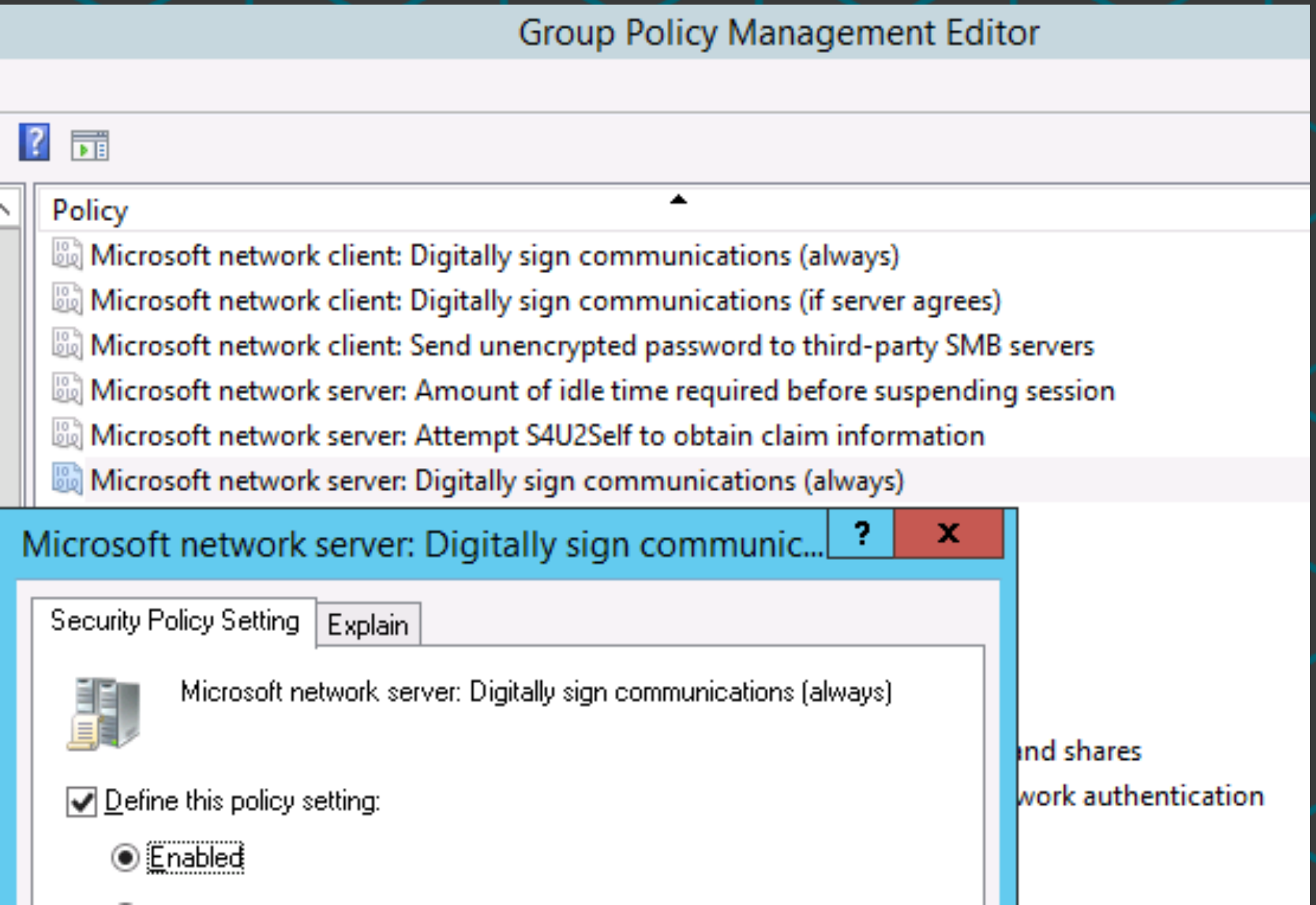
Github.com/QuickBreach/SMBetray.git

Countermeasures

> Countermeasures: Disable SMBv1



> Countermeasures: Require SMB Signing



Require SMB signatures across the domain on both clients **and** servers by enabling “Digitally sign communications (always)”

> Countermeasures: Require SMB Signing

> Countermeasures: Use Encryption

SMB 3 introduced support for encryption, which can be established on a per-share basis, or be implemented system-wide. Encryption can either be “supported”, or “required”.

If an SMB 3 server supports encryption, it will encrypt any session after authentication with an SMB 3 client, but not reject any un-encrypted connection with lesser versions of SMB

If an SMB 3 server requires encryption, it will reject any un-encrypted connection.

> Countermeasures: Use UNC Hardening

UNC hardening is a feature available in Group Policy that allows administrators to push connection security requirements to clients if the UNC the client is connecting to matches a certain pattern. These requirements include only supporting mutual authentication, requiring SMB signing, and requiring encryption.

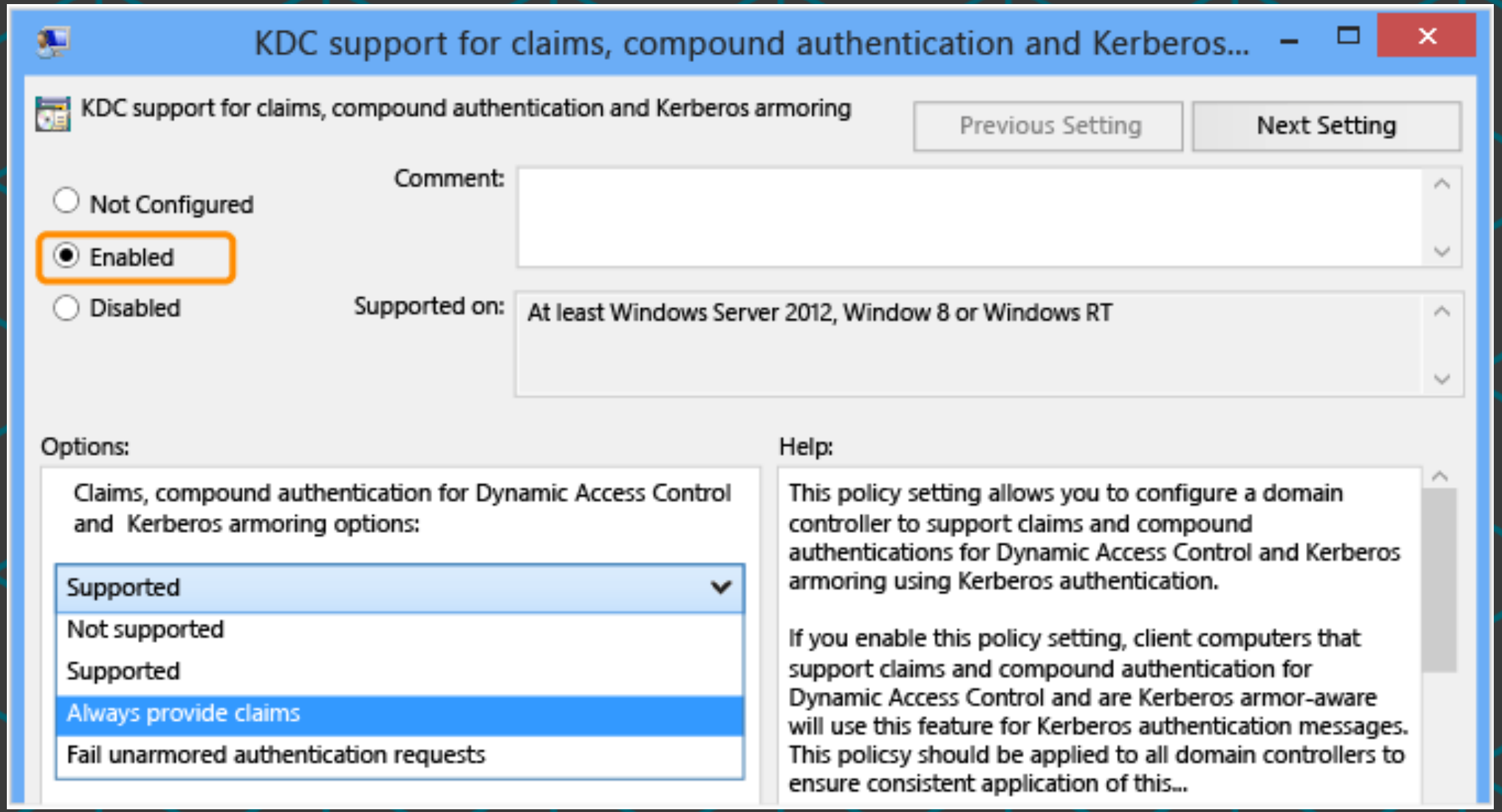
Eg. “RequireIntegrity=1” on “*\NETLOGON” will ensure that, regardless of the security requirements reported by the server, the client will only connect to the NETLOGON share if signing is used.

> Countermeasures: Audit & Restrict NTLM

Audit where NTLM is needed in your organization, and restrict it to those systems where it is needed.

Removing, or at least restricting NTLM in the environment, will aid in preventing authentication mechanism downgrades to NTLMv1/v2 for SMB dialects less than 3.1.1. The pre-authentication integrity hash in 3.1.1 protects its authentication mechanisms, but this check only occurs after authentication – which means an attacker would have already captured the NTLMv2 hashes to crack, even though the SMB 3.1.1 connection will be terminated.

> Countermeasures: Kerberos FAST Armoring



> Countermeasures: Kerberos FAST Armoring

Require Kerberos Flexible Authentication via Secure Tunneling (FAST)

- The user's Kerberos AS-REQ authentication is encapsulated within the machine's TGT. This prevents an attacker who knows the user's password from compromising that user's Kerberos session key. This also prevents attackers from cracking AS-REP's to compromise user passwords

- This feature enables authenticated Kerberos errors, preventing KDC error spoofing from downgrading clients to NTLM or weaker cryptography.

> Countermeasures: Kerberos FAST Armoring

PRECAUTIONS WHEN REQUIRING FAST:

Armoring requires Windows 8/2012, or later, throughout the environment. If FAST Armoring is required, and thereby set to “Fail unarmored authentication requests”, any older and non-FAST supporting devices will no longer be able to authenticate to the domain – and be dead in the water.

Review documentation & your environment thoroughly before requiring this setting

> Countermeasures: Passphrase not Password

Push users to the pass**phrase** from the pass**word** mindset

BAD: D3fc0n26!

GOOD: “5ome some stronger p@ssword!”

If we can’t crack the password in the first place, we can’t compromise Kerberos Session Keys or SMB Session Base Keys

> Contributors & Shoulders of Giants

> Contributors & Shoulders of Giants

Ned Pyle (@NerdPyle)
Principal Program Manager in the Microsoft Windows Server High Availability and Storage Group at Microsoft
- Verified authenticity of SMB protections and behaviors

Mathew George
Principal Software Engineer in the Windows Server Group at Microsoft
- Verified authenticity of SMB protections and behaviors

Special thanks to CoreSecurity for the impacket libraries

> \x00

Thank you DEFCON 26!

https://github.com/QuickBreach/SMBetray.git

William Martin
@QuickBreach