

Challenge 7 ("fullspeed")

Description

“Has this all been far too easy? Where's the math? Where's the science? Where's the, I don't know.... cryptography? Well we don't know about any of that, but here is a little .NET binary to chew on while you discuss career changes with your life coach.

Writeup

This challenge kicked my ass, but it was sooo worth it. Although every Flare-On challenge so far has taught me a lesson of its own, this one was by far the most valuable and multifaceted. Anyways, let's get into the writeup.

The zip contains a binary and a packet capture file. I opened the pcap in Wireshark and found a single TCP stream exchanged between 192.168.56.101 and 192.168.56.103.

The challenge description says that we will be analyzing a .NET binary. I still decided to open the binary in ExeinfoPE and Detect It Easy, neither of which seemed to recognize the sample as a .NET binary at all — both reported the compiler to be MSVC (C++). However, looking at the strings found in the sample by IDA, there were clear signs of .NET "presence" (e.g. `:BouncyCastle.Cryptography.dll, System.Private.CoreLib, 2System.Net.Primitives.dll$System.Net.Sockets`, etc.). Another interesting thing I noticed was the section headers. There were two sections that caught my eye: `.managed` and `hydrated`. Intrigued, I decided to google for this query: `"hydrated" section pe file`. What I found was a [blog post](#) from late 2023 titled "Reverse Engineering Natively Compiled .NET Apps". Reading it I understood why the aforementioned tools misclassified the compiler — ahead-of-time (AOT) compilation of .NET is a relatively recent feature and not (yet?) very widely used by legitimate software. The blog post also gives some great tips on finding the exact version of the compiled .NET runtime (in this case, 8.0.524.21615\8.0.5+087e15321bb712ef6fe8b0ba6f8bd12facf92629), as well as instructions on how to ahead-of-time compile your own .NET binary using the .NET Core CLI.

Reversing AOT compiled .NET

My first instinct (which turned out to be a really great idea) was to compile my own Hello World C# app and compare it to the sample. The structure of both binaries (including the `.managed` and `hydrated` sections) and the layout of both `main` functions was identical, so I knew this was exactly how the binary was created.

A couple google searches later, I found [another blog post](#) which talks more about concrete reversing techniques with IDA Pro, most notably generating custom FLIRT signature files and importing them into the database. I hadn't ever done this before, but I knew this was an absolutely crucial step towards being able to analyze the relevant parts of the binary instead of the .NET runtime itself. Just like the blog post author suggested, I asked an LLM to generate a large source file for me using as many classes and methods as possible, then generated the signatures and imported them into IDA, which to my great satisfaction made IDA recognize and name a vast majority of functions.

```

mov     [rsp+48h+var_20], rbp
mov     [rsp+48h+var_28], r10d
call    RhRegisterOSModule
test    al, al
jz      short _loc_140003322

```

```

lea     rdx, unk_7FF7427427B0
mov     [rsp+48h+var_28], 0Eh
lea     r8, unk_7FF7427427C0
mov     r9, rbp
sub     r8, rdx
mov     rcx, rsi
sar     r8, 3
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__InitializeModules
mov     rdx, rbx
mov     ecx, edi
call    __managed__Main
jmp     short _loc_140003327

```

```

_loc_14
mov

```

```

_loc_140003327:
mov     rbx, [rsp+48h+arg_0]
mov     rbp, [rsp+48h+arg_8]
mov     rsi, [rsp+48h+arg_10]
add     rsp, 40h
pop     rdi

```

The standard C `main` function sets up some things (for example, "rehydrates" the so called "dehydrated" data stored in the read-only data section) and calls `__managed__Main`, which again sets some things up and calls the user-defined module entry point, which I decided to call `UserMain`. Herein resides the actual logic of the program.

```

mov     rcx, [rax+50h]
test    rcx, rcx
jnz     short _loc_14013709C

```

```

call    S_P_CoreLib_System_Threading_Thread__InitializeCurrentThread
mov     rcx, rax

```

```

_loc_14013709C:
cmp     [rcx], c1
mov     edx, 1
mov     r8d, 1
call    S_P_CoreLib_System_Threading_Thread__SetApartmentState_0
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__RunModuleInitializers
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__GetMainMethodArguments
mov     rcx, rax
call    UserMain
call    sub_7FF74267FF40
call    S_P_CoreLib_System_AppContext__OnProcessExit
lea     rbx, unk_7FF742708C38
cmp     qword ptr [rbx-8], 0
jnz     short _loc_1401370ED

```

```

_loc_1401370ED:
call    sub_7FF7425B11D0

```

I also tried looking for RTTI information in the binary, but I didn't have any luck with that. Finding out that RTTI can be stripped from the binary during compilation, I decided not to investigate this further, as it simply wasn't worth it, and at this point it seemed I could pretty easily guess a lot of the basic types, such as strings, byte arrays, and `Span`s.

Analyzing the program logic

The logic of the `UserMain` function turned out to be surprisingly simple.

```
void UserMain(){           AppContext ctx = gAppContext; // (static variable)           string
addr = xorDecrypt(encryptedIpAddrAndPort) // 192.168.56.103;31337           [var ip, var portStr]
= addr.Split(xorDecrypt(encryptedSeparator)) // ;           var port = int.Parse(portStr);
ctx.tcpClient = new TcpClient(ip, port);           ctx.tcpStream = ctx.tcpClient.GetStream();
communicateOverTcp(); // name based on first look           doSomethingWithFileIo(); // ditto}
```

Based on the initial overview of the code, it seemed almost as if the application connected to and communicated with a C2 server of sorts, either sending files or receiving and executing commands.

I had considerable difficulty analyzing the `communicateOverTcp` function. This was due to a mistake I have made, which was that I did not pay enough attention to the strings mentioning BouncyCastle. BouncyCastle is a third party cryptographic library for the JVM and .NET ecosystems, and it certainly made sense that the TCP communication would be encrypted; however, for some reason, I initially didn't make the necessary distinction between the .NET standard cryptography library and BouncyCastle. I analyzed much of the `communicateOverTcp` function, found out that (probably) some version of Elliptic Curve Diffie-Hellman was used with a randomly chosen private key to establish a symmetric key (and nonce) that was then used with an (unauthenticated) stream cipher to encrypt communication. I analyzed the source of the randomness for the private key generation, the symmetric cipher and the high level outline of the key exchange (I knew it was ECDH thanks to some exception strings along the lines of "Invalid FpPoint coordinates" and the high level data flow), but I wasn't able to figure out where the weakness was. One thing struck me in the function that initialized what I called the `AppContext`: five different 48-byte BigIntegers were decrypted, constructed and used to initialize some of the other members, presumably the elliptic curve related objects. At this time, I wasn't sure if they could be parameters of the elliptic curve or perhaps a hardcoded private key. Finally, it occurred to me that this was way too difficult to analyze with so little information and that I must have been overseeing something. At this point I remembered seeing BouncyCastle among the strings and decided to create signatures for functions from this library. As it turned out, after doing this (I again asked an LLM to create a source code focusing on elliptic curves, ECDH, random number generation, KDFs and symmetric stream ciphers), basically no functions were left unrecognized by IDA. I felt kind of stupid, but at the same time, somewhat proud that I had the right idea about the key exchange being ECDH and was able to reverse engineer many of the BouncyCastle datatypes (such as `BigInteger`, `ECPoint`, partly `ChaChaEngine`) and their implementations.

Analyzing the key exchange and encryption scheme

Now armed with both more function signatures and more knowledge, I renamed `communicateOverTcp` to `ecdhEstablishKey` and began a more detailed analysis of the cryptographic aspects of the communication, especially the ECDH construction.

As it turned out, the five BigIntegers mentioned earlier were really (custom) EC parameters. The first number was the order of the \mathbb{F}_p field, the next two were the a and b parameters of the curve equation (recall that an elliptic curve over a finite field \mathbb{F}_p (with $p \geq 3$) and parameters a, b (which have to satisfy $4a^3 + 27b^2 \neq 0$) is nothing but the set of points $E = \{y^2 = x^3 + ax + b \mid x, y \in \mathbb{F}_p\} \cup \{\mathcal{O}\}$). The last couple of BigIntegers determined the (affine) coordinates of the "generator" point $G = (G_X, G_Y) \in E$.

This was certainly eye-catching, as the parameters didn't match any standard curve I was able to find (they certainly did not match any NIST curve). I suspected that this might be the focus of the challenge, but for certainty (and because I don't yet consider myself an expert on elliptic curve cryptography), I decided to verify the security of the rest of the scheme.

For the functions that matched my BouncyCastle signatures, I could now reasonably (though not for sure) make the assumption that they were not tweaked and indeed performed those

cryptographic operations that they are documented to perform. This way, I was able to quickly find out that the operation of the program could be summarized as follows:

1. Construct a (prime field) elliptic curve with custom parameters p, a, b and a generator point with coordinates G_X, G_Y .
2. Using the .NET `SecureRandom` class with an autoseeded SHA-256 CSPRNG provider from BouncyCastle, generate a 48-byte private key d .
3. Compute the coordinates of the public key $(P_X, P_Y) = P = d \cdot G$, XOR them with a salt (the 48-byte number `0x13371337...1337`) and send them to the C2 server.
4. Receive the C2 server's public key coordinates $(Q_X, Q_Y) = Q$, again XORed with the salt upon receipt.
5. Compute the shared point $(X_X, X_Y) = X = d \cdot Q$.
6. KDF: Take the SHA-512 hash of the 48-byte zero-padded X_X . The first 32 bytes shall be the symmetric key, the next 8 bytes shall be the nonce.
7. Initialize a ChaCha cipher with the computed key and nonce.
8. Use the ChaCha keystream to encrypt and decrypt all communication henceforth, avoiding any form of keystream reuse.

Attacking the elliptic curve

From these findings, it was clear that the only issue could lie in the usage of a custom curve. I used [SageCell](#) for a quick computation of the remaining parameters of the curve (i.e. order of the curve $|E|$, order of the generator point n , and the cofactor h) and searched for common attacks on improperly chosen curves. I learned that the cofactor should be small (ideally 1) and the order of a curve over \mathbb{F}_p should not equal p itself. Both of these conditions were met. Then I noticed an interesting phrase in the [Wikipedia entry on Elliptic curve cryptography](#) :

“For cryptographic application, the order of G [...] is normally prime.

But this wasn't the case with our curve — the order of G (equal to the order of E itself) could be factored into 8 prime factors. I decided to consult an LLM on this, asking what would happen if n wasn't prime. Helpfully (and correctly), the model pointed me to the [Pohlig-Hellman algorithm](#). Its idea took me a while to digest, but in the end, it seems rather simple:

0. The goal is to find the discrete logarithm of the public key Q with respect to G , i.e. find k such that $k \cdot G = Q$.
1. The order of G by definition is the smallest positive n such that $n \cdot G = \mathcal{O}$ (the identity element of the EC group). Since n could be factored into h_0, h_1, \dots, h_r (for some $r \geq 2$), the equation could be rephrased as $h_0 h_1 \dots h_r G = \mathcal{O}$.
2. Now, WLOG, consider the subgroup generated by $h_1 \dots h_r G$. Its order will be $n / (h_1 \dots h_r) = h_0$. Since h_0 is a relatively small number, it may be feasible to solve the discrete logarithm in this subgroup, i.e. find k_0 such that $k_0 \cdot G_0 = Q_0$, where $G_0 = \frac{n}{h_0} G$ and $Q_0 = \frac{n}{h_0} Q$, by brute force.
3. Finding the "reduced" discrete logarithm k_i for all $i \in \{0, \dots, r-1\}$, we get a system of congruences for the original k :

$$\begin{aligned} k \cdot h_1 h_2 \dots h_r &\equiv k_0 \cdot h_1 h_2 \dots h_r & (\text{mod } h_0 h_1 h_2 \dots h_r) \\ &\vdots \\ k \cdot h_0 h_1 \dots h_{r-1} &\equiv k_r \cdot h_0 h_1 \dots h_{r-1} & (\text{mod } h_0 h_1 h_2 \dots h_r) \end{aligned}$$

which can be rewritten as

$$k \equiv k_0 \pmod{h_0}$$

$$\vdots$$

$$k \equiv k_r \pmod{h_r}.$$

4. Since h_0, h_1, \dots, h_r are pairwise coprime (they are the unique prime factors of n , or more generally, their powers), this system of congruences can be solved using the simple version of the Chinese Remainder Theorem to get the unique $k \bmod n$.

Sadly, this algorithm could not really be applied in this form. While the factors h_0, \dots, h_6 were small and the ECDLPs brute-forceable in their corresponding subgroups, for the last factor, h_7 , this was not the case. The size of h_7 was about 2^{270} , so even using Pollard's Rho algorithm to solve the reduced DLP would take an order of about 2^{135} operations. Nonetheless, the idea of the algorithm could still technically be used. Leaving out the last factor in the system of congruences means that we will not get a unique solution for $k \bmod n$ by using CRT, but we can still reduce the keyspace by a factor of n/h_7 and hope that the challenge is constructed in a way that brute-forcing this (still unfeasibly huge) keyspace will find the solution in reasonable time.

As a matter of fact, I didn't think this approach had any chance at success (a specially crafted private key susceptible to a brute-force search seemed too artificial), so I only tested it very briefly (and as it turned out later, incorrectly) and began looking for other possible attacks and consulting my colleagues at work, some of which have a far better understanding of cryptography and elliptic curves than I do — I'd like to give them a shout-out for their massive help and willingness to share their expertise. Eventually, one of the colleagues who had already solved the challenge then hinted that my idea was indeed correct and would lead to the solution.

Retrieving the private key

Since I basically reverse engineered a whole part of the BouncyCastle library, I had a pretty good idea how it could be used to perform EC operations. Therefore, I chose to find the solution using C# (which I honestly never thought I would say).

I created an AOC .NET Core project, added the BouncyCastle dependency, and began work.

```
dotnet new console -o . --aotdotnet add package BouncyCastle.Cryptography
```

To retrieve the private key, I used the following code.

```
using Org.BouncyCastle.Crypto.Parameters;using Org.BouncyCastle.Math.EC;using BigInteger =
Org.BouncyCastle.Math.BigInteger;const int BRUTE_FORCE_ITER_COUNT =
1000000;CrackPrivateKeys();void CrackPrivateKeys(){    var ecParams =
GetEllipticCurveParams();    // The points we want to take the discrete logarithm of.    var
x_1 = new
BigInteger("195b46a760ed5a425dadcab37945867056d3e1a50124fffab78651193cea7758d4d590bed4f5f62d4a
291270f1dcf499", 16);    var y_1 = new
BigInteger("357731edebf0745d081033a668b58aaa51fa0b4fc02cd64c7e8668a016f0ec1317fcac24d8ec9f3e75
167077561e2a15", 16);    var x_2 = new
BigInteger("b3e5f89f04d49834de312110ae05f0649b3f0bbe2987304fc4ec2f46d6f036f1a897807c4e693e0bb5
cd9ac8a8005f06", 16);    var y_2 = new
BigInteger("85944d98396918741316cd0109929cb706af0cca1eaf378219c5286bdc21e979210390573e3047645e
1969bdbcb667eb", 16);    var q_1 = ecParams.Curve.CreatePoint(x_1, y_1);    var q_2 =
ecParams.Curve.CreatePoint(x_2, y_2);    // The divisors of the curve (or generator) order, n.
var ps = new BigInteger[] {        BigInteger.ValueOf(35809),
BigInteger.ValueOf(46027),        BigInteger.ValueOf(56369),        BigInteger.ValueOf(57301),
BigInteger.ValueOf(65063),        BigInteger.ValueOf(111659),
BigInteger.ValueOf(113111),        // new
```

```

BigInteger("7072010737074051173701300310820071551428959987622994965153676442076542799542912293", 10),    };    //var ks_1 = SolvePartialEcdLps(q_1, g, n, ps, ecParams);
//Console.WriteLine($"Partial ECDLPs for Q_1: ");    //PrintArray(ks_1);    //Partial ECDLPs for
Q_1: [11872, 42485, 12334, 45941, 27946, 43080, 57712]    var ks_1 = new BigInteger[] {
BigInteger.ValueOf(11872), BigInteger.ValueOf(42485), BigInteger.ValueOf(12334),
BigInteger.ValueOf(45941), BigInteger.ValueOf(27946), BigInteger.ValueOf(43080),
BigInteger.ValueOf(57712) };    //var ks_2 = SolvePartialEcdLps(q_2, g, n, ps, ecParams);
//Console.WriteLine($"Partial ECDLPs for Q_2: ");    //PrintArray(ks_2);    //Partial ECDLPs for
Q_2: [26132, 27202, 25870, 52801, 26868, 60997, 95883]    var ks_2 = new BigInteger[] {
BigInteger.ValueOf(26132), BigInteger.ValueOf(27202), BigInteger.ValueOf(25870),
BigInteger.ValueOf(52801), BigInteger.ValueOf(26868), BigInteger.ValueOf(60997),
BigInteger.ValueOf(95883) };    var (k_1, step_1) = Crt(ps, ks_1);    VerifyCrtResult(ps,
ks_1, k_1);    Console.WriteLine($"CRT result for Q_1: k_0 = {k_1}, step = {step_1}");    var
(k_2, step_2) = Crt(ps, ks_2);    VerifyCrtResult(ps, ks_2, k_2);    Console.WriteLine($"CRT
result for Q_2: k_0 = {k_2}, step = {step_2}");    //if (g.Multiply(k_1).Equals(q_1))    //
Console.WriteLine($"PRIVATE KEY FOUND FOR Q_1: {k_1}");    //if (g.Multiply(k_2).Equals(q_2))
//    Console.WriteLine($"PRIVATE KEY FOUND FOR Q_2: {k_2}");    for (int i = 0; i <
ps.Length; i++)    {        var q_i_1 = q_1.Multiply(ecParams.N.Divide(ps[i]));        var
q_i_2 = q_2.Multiply(ecParams.N.Divide(ps[i]));        var g_i =
ecParams.G.Multiply(ecParams.N.Divide(ps[i]));
Assert(g_i.Multiply(ks_1[i]).Equals(q_i_1), $"The discrete logarithm for Q_1 and i={i} is not
correct.");    Assert(g_i.Multiply(ks_2[i]).Equals(q_i_2), $"The discrete logarithm for
Q_2 and i={i} is not correct.");    }    Assert(step_1.Equals(step_2), "The steps are not
equal.");    var step_multiple_of_g = ecParams.G.Multiply(step_1);    var current_1 =
ecParams.G.Multiply(k_1);    var current_2 = ecParams.G.Multiply(k_2);    for (int i = 0; i <
BRUTE_FORCE_ITER_COUNT; i++)    {        if (current_1.Equals(q_1))        {
Console.WriteLine($"PRIVATE KEY FOUND FOR Q_1: {k_1.ToString(16)}");        return;
}        if (current_2.Equals(q_2))        {        Console.WriteLine($"PRIVATE KEY FOUND
FOR Q_2: {k_2.ToString(16)}");        return;        }        current_1 =
current_1.Add(step_multiple_of_g);        current_2 = current_2.Add(step_multiple_of_g);
k_1 = k_1.Add(step_1);        k_2 = k_2.Add(step_2);    }    Console.WriteLine($"Neither
private key is found after {BRUTE_FORCE_ITER_COUNT} iterations.");}void Assert(bool condition,
string message){    if (!condition) throw new InvalidOperationException(message);}void
PrintArray<T>(T[] array){    Console.Write("[");    for (int i = 0; i < array.Length; i++)
{        Console.Write(array[i]);        if (i < array.Length - 1)            Console.Write(",
");    }    Console.WriteLine("]");}ECDomainParameters GetEllipticCurveParams(){    // Curve
domain parameters.    var p = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E5576A7A56976C2BAECA47809765283AA0785
83E1E65172A3FD", 16);    var a = new
BigInteger("A079DB08EA2470350C182487B50F7707DD46A58A1D160FF79297DCC9BFAD6CFC96A81C4A97564118A4
0331FE0FC1327F", 16);    var b = new
BigInteger("9F939C02A7BD7FC263A4CCE416F4C575F28D0C1315C4F0C282FCA6709A5F9F7F9C251C9EED9EB1BAA
31602167FA5380", 16);    var g_X = new
BigInteger("087B5FE3AE6DCFB0E074B40F6208C8F6DE4F4F0679D6933796D3B9BD659704FB85452F041FFF14CF0E
9AA7E45544F9D8", 16);    var g_Y = new
BigInteger("127425C1D330ED537663E87459EAA1B1B53EDFE305F6A79B184B3180033AAB190EB9AA003E02E9DBF6
D593C5E3B08182", 16);    var n = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E547761EC3EA549979D50C95478998110005C
8C2B7F3498EE71", 16);    // Create the curve and generator point.    var curve = new
FpCurve(p, a, b);    var g = curve.CreatePoint(g_X, g_Y);    var ecParams = new
ECDomainParameters(curve, g, n, BigInteger.One);    return ecParams;}// Chinese Remainder
Theorem (CRT) for solving the system of linear congruences.// Returns the solution x and the
modulus n.(BigInteger, BigInteger) Crt(BigInteger[] ps, BigInteger[] ks){    var n =

```

```

BigInteger.One;    for (int i = 0; i < ps.Length; i++)        n = n.Multiply(ps[i]);    var ns
= new BigInteger[ps.Length];    var ms = new BigInteger[ps.Length];    var ys = new
BigInteger[ps.Length];    for (int i = 0; i < ps.Length; i++)    {        ns[i] =
n.Divide(ps[i]);        ms[i] = ns[i].ModInverse(ps[i]);        ys[i] =
ns[i].Multiply(ms[i]).Mod(n);    }    var x = BigInteger.Zero;    for (int i = 0; i <
ps.Length; i++)        x = x.Add(ks[i].Multiply(ys[i]));    return (x.Mod(n), n);}void
VerifyCrtResult(BigInteger[] ps, BigInteger[] ks, BigInteger x){    for (int i = 0; i <
ps.Length; i++)        Assert(x.Mod(ps[i]).Equals(ks[i]), $"The CRT result is not correct for
i={i}.");}BigInteger[] SolvePartialEcdlps(ECPoint q, ECPoint g, BigInteger n, BigInteger[] ps,
ECDomainParameters ecParams){    var ks = new BigInteger[ps.Length];    for (int i = 0; i <
ps.Length; i++)    {        var q_i = q.Multiply(n.Divide(ps[i]));        var g_i =
g.Multiply(n.Divide(ps[i]));        ks[i] = Ecdlp(q_i, g_i, ecParams);
//Console.WriteLine($"[{i}/{ks.Length}]: p_i = {ps[i]} ==> k_i = {ks[i]}");    }    return
ks;}// Brute force discrete logarithm solver for a point q = k * g.BigInteger Ecdlp(ECPoint q,
ECPoint g, ECDomainParameters ecParams){    var x = g;    var k = BigInteger.One;    while
(true)    {        if (k.CompareTo(ecParams.N) == 0)            throw new
InvalidOperationException("The discrete logarithm is not found.");        if (x.Equals(q))
return k;        x = x.Add(g);        k = k.Add(BigInteger.One);    }}

```

With the "reduced" ECDLPs pre-computed, this only takes maybe three seconds to finish, and it indeed finds the private key of the C2 server:

```

CRT result for Q_1: k_0 = 3914004671535485983675163411331184, step =
4374617177662805965808447230529629CRT result for Q_2: k_0 =
1347455424744677257745571369218247, step = 4374617177662805965808447230529629PRIVATE KEY FOUND
FOR Q_2: 73a3e816c7642f57e6bd4c6079a19d64

```

Decrypting the flag

This means we can now derive the symmetric key and decrypt the packets from the capture file! Again, I chose to do this in C#, for the same reasons stated above.

```

using Org.BouncyCastle.Crypto.Engines;using Org.BouncyCastle.Crypto.Parameters;using
Org.BouncyCastle.Math.EC;using BigInteger = Org.BouncyCastle.Math.BigInteger;byte[][] PACKETS
= [    [0xf2, 0x72, 0xd5, 0x4c, 0x31, 0x86, 0x0f],    [0x3f, 0xbd, 0x43, 0xda, 0x3e, 0xe3,
0x25],    [0x86, 0xdf, 0xd7],    [0xc5, 0x0c, 0xea, 0x1c, 0x4a, 0xa0, 0x64, 0xc3, 0x5a, 0x7f,
0x6e, 0x3a, 0xb0, 0x25, 0x84, 0x41, 0xac, 0x15, 0x85, 0xc3, 0x62, 0x56, 0xde, 0xa8, 0x3c,
0xac, 0x93, 0x00, 0x7a, 0x0c, 0x3a, 0x29, 0x86, 0x4f, 0x8e, 0x28, 0x5f, 0xfa, 0x79, 0xc8,
0xeb, 0x43, 0x97, 0x6d, 0x5b, 0x58, 0x7f, 0x8f, 0x35, 0xe6, 0x99, 0x54, 0x71, 0x16],    [0xfc,
0xb1, 0xd2, 0xcd, 0xbb, 0xa9, 0x79, 0xc9, 0x89, 0x99, 0x8c],    [0x61, 0x49, 0x0b],    [0xce,
0x39, 0xda],    [0x57, 0x70, 0x11, 0xe0, 0xd7, 0x6e, 0xc8, 0xeb, 0x0b, 0x82, 0x59, 0x33, 0x1d,
0xef, 0x13, 0xee, 0x6d, 0x86, 0x72, 0x3e, 0xac, 0x9f, 0x04, 0x28, 0x92, 0x4e, 0xe7, 0xf8,
0x41, 0x1d, 0x4c, 0x70, 0x1b, 0x4d, 0x9e, 0x2b, 0x37, 0x93, 0xf6, 0x11, 0x7d, 0xd3, 0x0d,
0xac, 0xba],    [0x2c, 0xae, 0x60, 0x0b, 0x5f, 0x32, 0xce, 0xa1, 0x93, 0xe0, 0xde, 0x63, 0xd7,
0x09, 0x83, 0x8b, 0xd6],    [0xa7, 0xfd, 0x35],    [0xed, 0xf0, 0xfc],    [0x80, 0x2b, 0x15,
0x18, 0x6c, 0x7a, 0x1b, 0x1a, 0x47, 0x5d, 0xaf, 0x94, 0xae, 0x40, 0xf6, 0xbb, 0x81, 0xaf,
0xce, 0xdc, 0x4a, 0xfb, 0x15, 0x8a, 0x51, 0x28, 0xc2, 0x8c, 0x91, 0xcd, 0x7a, 0x88, 0x57,
0xd1, 0x2a, 0x66, 0x1a, 0xca, 0xec],    [0xae, 0xc8, 0xd2, 0x7a, 0x7c, 0xf2, 0x6a, 0x17, 0x27,
0x36, 0x85],    [0x35, 0xa4, 0x4e],    [0x2f, 0x39, 0x17],    [0xed, 0x09, 0x44, 0x7d, 0xed,
0x79, 0x72, 0x19, 0xc9, 0x66, 0xef, 0x3d, 0xd5, 0x70, 0x5a, 0x3c, 0x32, 0xbd, 0xb1, 0x71,
0x0a, 0xe3, 0xb8, 0x7f, 0xe6, 0x66, 0x69, 0xe0, 0xb4, 0x64, 0x6f, 0xc4, 0x16, 0xc3, 0x99,
0xc3, 0xa4, 0xfe, 0x1e, 0xdc, 0x0a, 0x3e, 0xc5, 0x82, 0x7b, 0x84, 0xdb, 0x5a, 0x79, 0xb8,
0x16, 0x34, 0xe7, 0xc3, 0xaf, 0xe5, 0x28, 0xa4, 0xda, 0x15, 0x45, 0x7b, 0x63, 0x78, 0x15,

```

```

0x37, 0x3d, 0x4e, 0xdc, 0xac, 0x21, 0x59, 0xd0, 0x56],    [0xf5, 0x98, 0x1f, 0x71, 0xc7, 0xea,
0x1b, 0x5d, 0x8b, 0x1e, 0x5f, 0x06, 0xfc, 0x83, 0xb1, 0xde, 0xf3, 0x8c, 0x6f, 0x4e, 0x69,
0x4e, 0x37, 0x06, 0x41, 0x2e, 0xab, 0xf5, 0x4e, 0x3b, 0x6f, 0x4d, 0x19, 0xe8, 0xef, 0x46,
0xb0, 0x4e, 0x39, 0x9f, 0x2c, 0x8e, 0xce, 0x84, 0x17, 0xfa],    [0x40, 0x08, 0xbc],    [0x54,
0xe4, 0x1e],    [0xf7, 0x01, 0xfe, 0xe7, 0x4e, 0x80, 0xe8, 0xdf, 0xb5, 0x4b, 0x48, 0x7f, 0x9b,
0x2e, 0x3a, 0x27, 0x7f, 0xa2, 0x89, 0xcf, 0x6c, 0xb8, 0xdf, 0x98, 0x6c, 0xdd, 0x38, 0x7e,
0x34, 0x2a, 0xc9, 0xf5, 0x28, 0x6d, 0xa1, 0x1c, 0xa2, 0x78, 0x40, 0x84],    [0x5c, 0xa6, 0x8d,
0x13, 0x94, 0xbe, 0x2a, 0x4d, 0x3d, 0x4d, 0x7c, 0x82, 0xe5],    [0x31, 0xb6, 0xda, 0xc6, 0x2e,
0xf1, 0xad, 0x8d, 0xc1, 0xf6, 0x0b, 0x79, 0x26, 0x5e, 0xd0, 0xde, 0xaa, 0x31, 0xdd, 0xd2,
0xd5, 0x3a, 0xa9, 0xfd, 0x93, 0x43, 0x46, 0x38, 0x10, 0xf3, 0xe2, 0x23, 0x24, 0x06, 0x36,
0x6b, 0x48, 0x41, 0x53, 0x33, 0xd4, 0xb8, 0xac, 0x33, 0x6d, 0x40, 0x86, 0xef, 0xa0, 0xf1,
0x5e, 0x6e, 0x59],    [0x0d, 0x1e, 0xc0, 0x6f, 0x36],];DecryptTCP(PACKETS);void
DecryptTCP(byte[][] packets){    var ec = GetEllipticCurveParams();    var cncPrivateKey = new
BigInteger("73a3e816c7642f57e6bd4c6079a19d64", 16);    var bdPublicKey = ec.Curve.CreatePoint(
new
BigInteger("195b46a760ed5a425dadcab37945867056d3e1a50124fffab78651193cea7758d4d590bed4f5f62d4a
291270f1dcf499", 16),    new
BigInteger("357731edebf0745d081033a668b58aaa51fa0b4fc02cd64c7e8668a016f0ec1317fcac24d8ec9f3e75
167077561e2a15", 16)    );    var sharedSecret =
bdPublicKey.Multiply(cncPrivateKey).Normalize().XCoord.ToBigInteger();    var keyMaterial =
Kdf(sharedSecret, ec.N.BitLength);    var key = new byte[32];    Array.Copy(keyMaterial, 0,
key, 0, key.Length);    Console.WriteLine("Using key: " + Hex(key));    var nonce = new
byte[8];    Array.Copy(keyMaterial, key.Length, nonce, 0, nonce.Length);
Console.WriteLine("Using nonce: " + Hex(nonce));    var engine = new ChaChaEngine(20);
engine.Init(true /* doesn't matter */, new ParametersWithIV(new KeyParameter(key), nonce));
DecryptAndPrintPackets(packets, engine);}ECDomainParameters GetEllipticCurveParams(){    //
Curve domain parameters.    var p = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E5576A7A56976C2BAECA47809765283AA0785
83E1E65172A3FD", 16);    var a = new
BigInteger("A079DB08EA2470350C182487B50F7707DD46A58A1D160FF79297DCC9BFAD6CFC96A81C4A97564118A4
0331FE0FC1327F", 16);    var b = new
BigInteger("9F939C02A7BD7FC263A4CCE416F4C575F28D0C1315C4F0C282FCA6709A5F9F7F9C251C9EEDE9EB1BAA
31602167FA5380", 16);    var g_X = new
BigInteger("087B5FE3AE6DCFB0E074B40F6208C8F6DE4F4F0679D6933796D3B9BD659704FB85452F041FFF14CF0E
9AA7E45544F9D8", 16);    var g_Y = new
BigInteger("127425C1D330ED537663E87459EAA1B1B53EDFE305F6A79B184B3180033AAB190EB9AA003E02E9DBF6
D593C5E3B08182", 16);    var n = new
BigInteger("C90102FAA48F18B5EAC1F76BB40A1B9FB0D841712BBE3E547761EC3EA549979D50C95478998110005C
8C2B7F3498EE71", 16);    // Create the curve and generator point.    var curve = new
FpCurve(p, a, b);    var g = curve.CreatePoint(g_X, g_Y);    var ecParams = new
ECDomainParameters(curve, g, n, BigInteger.One);    return ecParams;}string Hex(byte[] bytes){
var sb = new System.Text.StringBuilder(bytes.Length * 2);    foreach (var b in bytes)
sb.Append(b.ToString("X2"));    return sb.ToString();}void DecryptAndPrintPackets(byte[][]
packets, ChaChaEngine engine){    int i = 0;    foreach (var packet in packets)    {        if
(i++ > 0) Console.WriteLine();        var decrypted = new byte[packet.Length];
engine.ProcessBytes(packet, 0, packet.Length, decrypted, 0);
Console.WriteLine(System.Text.Encoding.ASCII.GetString(decrypted));    }byte[] Kdf(BigInteger
ecdhResult, int bitLength){    using (var sha512 =
System.Security.Cryptography.SHA512.Create())    {        byte[] buffer = new byte[bitLength /
8];        ecdhResult.ToByteArrayUnsigned(buffer.AsSpan());        return
sha512.ComputeHash(buffer);    }}

```

Output:

Using key: B48F8FA4C856D496ACDECD16D9C94CC6B01AA1C0065B023BE97AFDD12156F3DCUsing nonce:
3FD480978485D818(...)cat|flag.txtRDBudF9VNWVfeTB1cl9Pd25fQ3VSdjNzQGZsYXJLLW9uLmNvbQ==exit

Decoding the flag from base64 produces `D0nt_U5e_y0ur_0wn_CuRv3s@flare-on.com`.

🕒 Revision #8

★ Created 23 December 2024 21:40:17 by Annatar

✎ Updated 24 December 2024 00:09:30 by Annatar