ECDH Write Up - Red Alert ICS CTF

Summary

The Red Alert ICS CTF is an annual CTF held by NSHC Security at DEF CON. During DEF CON 31, our team won the CTF, earning a DEF CON Black Badge.

The CTF had a challenge titled ECDH. The challenge prompt was as follows:

> You have obtained the critical information from the cyber vault for controlling the crane. But the problem is that the file is encrypted. Decrypt the file and capture the flag.

The challenge requires the player to learn about the ECDH protocol and gain familiarity with the PyCryptodome and cryptography Python libraries. The player is given three files (see top left for downloading these files yourself):

- chal.zip.enc: Encrypted zip file containing the flag
- ecdh.py: The code that was used to encrypt the file
- result.txt: Command line output of the encryption process

`ecdh.py` contains a private key while `result.txt` contains a public key. These keys are used to generate a shared secret that is ultimately used with AES to encrypt a file. By knowing the public key of one party and the private key of another, it is trivial to regenerate that same exact shared secret and decrypt the file. The player needs to gain the understanding of ECDH and how it works in theory, then read the documentation for the cryptography library to learn how to import a public key from hex characters and perform the key exchange.

Details

Explaining ecdh.py

From reading the import statements, we can see that AES is imported from PyCryptodome on line 5, and ECDH is imported on line 8.

```
#!/usr/bin/env python3
import hashlib, hmac
import binascii, sys, struct, os
from hexdump import hexdump
from Crypto.Cipher import AES
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, padding, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import PublicFormat
from cryptography.hazmat.primitives.serialization import Encoding
```

The functions `encrypt_file` and `decrypt_file` are overly-complicated AES-CBC encryption and decryption routines, so we can safely ignore those.

We can then see the "define" section of the code, denoted with a comment. This section begins by defining a curve, a signature algorithm, a salt, and an HKDF function:

```
# define ----------------------------------------------------------------|
curve = ec.SECP256R1()
signature_algorithm = ec.ECDSA(hashes.SHA256())

salt=b'MOTIECTF-MessageKeys'
hkdf = HKDF(
    algorithm=hashes.SHA256(),
    length=0x30,
    salt=salt,
    info=None,
    backend=default_backend()
)
```

These are all, more or less, irrelevant. We will go over why later. For now, these variables should be left alone.

Next, a private key is generated using the above-defined curve, and its public component is dumped to the screen:

```
ephemeralKey     = ec.generate_private_key(curve, default_backend())   # ephemeral_private
ephemeralPubKey  = ephemeralKey.public_key()
print("[+] ephemeralPubKey ---")
hexdump(ephemeralPubKey.public_bytes(Encoding.DER, PublicFormat.SubjectPublicKeyInfo))
```

The `ephemeralKey` is *never* printed. Because it is defined from a curve, it will not be possible to regenerate this same key (something something heat death of the universe). The public key, however, is available in the result.txt file:

```
[+] ephemeralPubKey ---
00000000: 30 59 30 13 06 07 2A 86  48 CE 3D 02 01 06 08 2A  0Y0...*.H.=....*
00000010: 86 48 CE 3D 03 01 07 03  42 00 04 1E CF FB A9 9B  .H.=....B.......
00000020: A9 69 9A 73 BA 89 AB 9B  8B 1C 3F 98 9E 77 2A CD  .i.s......?..w*.
00000030: 6D 6A 1B 40 CB 4C 8F 7C  2A 14 43 99 10 DA B0 3F  mj.@.L.|*.C....?
00000040: 0D 87 A9 0D 83 D2 41 11  BF 5A 81 51 85 44 D6 F6  ......A..Z.Q.D..
00000050: 5C FE 54 7F DC 3F E4 E1  A5 66 D8                \.T..?...f.
```

The next three lines are crucial. These sections define a private key; not from a curve, but from raw bytes:

```
privKey     = int.from_bytes(bytes.fromhex("5a55034a6c8ce32e efc745faf7e5e2a8 d24cadd2116ab132
8b634f21f6b21706"), "big")
privKey     = ec.derive_private_key(privKey, curve, default_backend())
pubKey      = privKey.public_key()
```

Next, the shared secret is created, and the AES key and IV are derived from that shared secret using the above-defined HKDF algorithm:

```
shared_key  = ephemeralKey.exchange(ec.ECDH(), pubKey)
derived_key = hkdf.derive(shared_key)
key         = derived_key[:0x20]
iv          = derived_key[0x20:]
```

What does this mean?

The code generates a private key called `ephemeralKey` that is eventually garbage-collected and lost. It also uses another hard-coded private key to generate a public key. That public key is then used with `ephemeralKey` to create what eventually becomes the AES key and IV.

The file is encrypted with AES-CBC using the variable `derived_key` as its key and IV. We want to generate the same `derived_key`. We know that this variable is created via `hkdf.derive(shared_key)`. We

don't care what HKDF is or what it does. It only takes in one variable, so it stands to reason if we can generate that variable, we can generate `derived_key` via the same exact function call. HKDF takes in `shared_key`, so our ultimate goal is to create the same `shared_key` variable and call the `decrypt_file` function.

Let's break down the call that defines `shared_key`. The function is `ephemeralKey.exchange`. `ephemeralKey` is created using `ec.generate_private_key`. `ec` is imported from `cryptography.hazmat.primitives.asymmetric`. Time to read some documentation for that import, available here:

Elliptic curve cryptography

`cryptography.hazmat.primitives.asymmetric.ec.generate_private_key`(*curve*) [source]

New in version 0.5.

Generate a new private key on `curve`.

Parameters: **curve** – An instance of `EllipticCurve`.

Returns: A new instance of `EllipticCurvePrivateKey`.

We see that the return of `ec.generate_private_key` is a "new instance of `EllipticCurvePrivateKey`." Clicking on that, we see at the very top the same `exchange` function we saw before in `shared_key = ephemeralKey.exchange`:

class **cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey** [source]

New in version 0.5.

An elliptic curve private key for use with an algorithm such as ECDSA.

exchange(*algorithm, peer_public_key*) [source]

New in version 1.1.

Performs a key exchange operation using the provided algorithm with the peer's public key.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Parameters:
- **algorithm** – The key exchange algorithm, currently only `ECDH` is supported.
- **peer_public_key** (*EllipticCurvePublicKey*) – The public key for the peer.

Returns bytes: A shared key.

The two variables are a public key and any key exchange algorithm, as long as it's ECDH. For some reason, this makes me think of cars and their varied colors, not sure why though. Anyway, we know that the function call uses ECDH, as the full line is `shared_key = ephemeralKey.exchange(ec.ECDH(), pubKey)`. Clicking on ECDH, we get a big wall of text and code. We're close.

class **cryptography.hazmat.primitives.asymmetric.ec.ECDH** [source]

New in version 1.1.

The Elliptic Curve Diffie-Hellman Key Exchange algorithm standardized in NIST publication 800-56A.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Note that while elliptic curve keys can be used for both signing and key exchange, this is bad cryptographic practice. Instead, users should generate separate signing and ECDH keys.

We now know that the AES-CBC key is created using "the Elliptic Curve Diffie-Hellman Key Exchange algorithm standardized in NIST publication 800-56A." Now, as much as I love NIST, I am not reading 152 pages during a CTF. Similarly, the Wikipedia page looks equally as scary as the NIST paper—there are far too many weird-looking Greek letters. I need someone to explain to me what ECDH does, do so using layman terms, and answer any of my follow up questions quickly, without any snark, without asking me what I'm trying to do, without telling me to do something different, without asking me to take my comments to Stack Overflow chat, and with the knowledge

that can only be created by scraping the entire internet.

Default (GPT-3.5)

HE How does ECDH work in simple terms?

At its core, ECDH (Elliptic Curve Diffie-Hellman) is a cryptographic method that allows two parties to securely exchange a secret key over an insecure communication channel. This secret key can then be used for encrypting and decrypting messages between these parties.

Here's how ECDH works in simple terms:

Oh, OpenAI, how I love you. Our friend, ChatGPT, gives us a clear answer. The most important part is the following:

3. **Key Exchange**: When both parties have their own private and public keys, they exchange their public keys over the insecure channel.
4. **Shared Secret Calculation**: Using the received public key from the other party and their own private key, each party performs another mathematical operation. This operation results in a shared secret that is the same for both parties, even though they have different private keys.

We have the hard-coded private key and the ephemeral public key. All we need to do is call the same functions but with different keys, as the "operation results in a shared secret that is the same for both parties, even though they have different private keys."

Solution:

We just need to make a few adjustments. We replace the `ephemeralPubKey` with the public key from `result.txt`:

```
ephemeralPubKey = serialization.load_der_public_key(bytes.fromhex("3059301306072A86 48CE3D020106082A 8648CE3D03010703 4200041ECFFBA99B A9699A73BA89AB9B 8B1C3F989E772ACD 6D6A1B40CB4C8F7C 2A14439910DAB03F 0D87A90D83D24111 BF5A81518544D6F6 5CFE547FDC3FE4E1 A566D8"))
```

Instead of creating the shared_key using the ephemeralKey and pubKey, we instead use privKey and ephemeralPubKey:

```
shared_key = privKey.exchange(ec.ECDH(), ephemeralPubKey)
```

We update the file name:

```
filename    = 'chal.zip.enc'
```

And we decrypt, instead of encrypting:

```
decrypt_file(key, iv, filename)
```

Unzipping the file we get the flag `RACTF{Elliptic_curv3_Diffi3_H3llm4n!!}`.

Full Solution:

```
#!/usr/bin/env python3
import hashlib, hmac
import binascii, sys, struct, os
from hexdump import hexdump
from Crypto.Cipher import AES
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, padding, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import PublicFormat
from cryptography.hazmat.primitives.serialization import Encoding

def encrypt_file(key, iv, in_filename, out_filename=None, chunksize=64*1024):
    if not out_filename:
        out_filename = in_filename + '.enc'

    encryptor = AES.new(key, AES.MODE_CBC, iv)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            outfile.write(struct.pack('<Q', filesize))

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)
```

```
            outfile.write(encryptor.encrypt(chunk))

def decrypt_file(key, iv, in_filename, out_filename=None, chunksize=24*1024):
    if not out_filename:
        out_filename = os.path.splitext(in_filename)[0]

    with open(in_filename, 'rb') as infile:
        origsize = struct.unpack('<Q', infile.read(struct.calcsize('Q')))[0]
        decryptor = AES.new(key, AES.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                outfile.write(decryptor.decrypt(chunk))

            outfile.truncate(origsize)

# define ---------------------------------------------------------------|
curve = ec.SECP256R1()
signature_algorithm = ec.ECDSA(hashes.SHA256())

salt=b'MOTIECTF-MessageKeys'
hkdf = HKDF(
    algorithm=hashes.SHA256(),
    length=0x30,
    salt=salt,
    info=None,
    backend=default_backend()
)

#ephemeralKey     = ec.generate_private_key(curve, default_backend())   # ephemeral_private
ephemeralPubKey  = serialization.load_der_public_key(bytes.fromhex("3059301306072A86 48CE3D020106082A
8648CE3D03010703 4200041ECFFBA99B A9699A73BA89AB9B 8B1C3F989E772ACD 6D6A1B40CB4C8F7C
2A14439910DAB03F 0D87A90D83D24111 BF5A81518544D6F6 5CFE547FDC3FE4E1 A566D8"))
print("[+] ephemeralPubKey ---")
hexdump(ephemeralPubKey.public_bytes(Encoding.DER, PublicFormat.SubjectPublicKeyInfo))
```

```
privKey     = int.from_bytes(bytes.fromhex("5a55034a6c8ce32e efc745faf7e5e2a8 d24cadd2116ab132
8b634f21f6b21706"), "big")
privKey     = ec.derive_private_key(privKey, curve, default_backend())
pubKey      = privKey.public_key()

shared_key  = privKey.exchange(ec.ECDH(), ephemeralPubKey)
derived_key = hkdf.derive(shared_key)
key         = derived_key[:0x20]
iv          = derived_key[0x20:]

# Encrypting file
print("[+] Decrypting file")
filename    = 'chal.zip.enc'
decrypt_file(key, iv, filename)

print("[+] Decryption DONE")
```

Licensing

This page is licensed under a Creative Commons Attribution 4.0 International License. All code snippets within this page are licensed under a Creative Commons Universal (CC0 1.0) Public Domain Dedication.

Revision #4
Created 18 August 2023 04:05:05 by Henry Reed
Updated 18 August 2023 07:44:13 by Henry Reed