All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them.

Crypto Attacks

STREAM CIPHER REUSE

Looking back at stream ciphers, we see that we generate a stream of random bytes which are XORed with the plaintext to produce cipher text.
Anytime you feed a given key into a stream cipher, it's always going to generate the same output bytes. This is expected, and it's why decryption works exactly the same as encryption.

But let's consider the case of two pieces of data (A and B) being encrypted with the same key (K, generating stream S).

A_enc[0] = S[0] ^ A[0]
B_enc[0] = S[0] ^ B[0]
A_enc[0] = S[0] ^ A[0]
B_enc[0] = S[0] ^ B[0]

Because of the fact that both A and B are being XORed with the same random stream, it is trivial to unroll this operation.

(A[0] ^ S[0]) ^ (B[0] ^ S[0]) == A[0] ^ B[0]

Thus: A_enc[0] ^ B_enc[0] == A[0] ^ B[0]

By XORing the two cipher texts together, we get the XOR of the plaintext – the random stream cancels out entirely!

This is not a problem for modern stream ciphers like eSTREAM, which take a nonce along with the key. This means that as long as you don't reuse a given key-nonce pair, you're safe. However, the most common stream cipher you'll see is RC4 key be XORed with a nonce prior to encryption/decryption. This is not perfect, but it mitigates the issue.

ECB BLOCK RENDERING

Since we know that blocks encrypted with ECB are independent, there's nothing stopping us from arbitrarily reordering them; they'll still decrypt properly if this happens.
So consider a DES-ECB-encrypted cookie containing the following data:
admin=0;username=daeken

Because DES uses 8-byte blocks, this mean we'd encrypt the following blocks (with @ as padding – we'll talk about that part later):

admin=0;
username
=daeken@

But let’s say we control our username. We don’t need to be able to encrypt or decrypt data to make ourselves an admin. If we set our username to paddingadmin=1; we get these blocks:

admin=0;
username
=padding
admin=1;

Once we have the encrypted form of this new cookie, we simply take the last block and put it in a place of the first. Suddenly, this decrypts to a valid admin cookie!

This attack is absolutely trivial and inherent to ECB.

ECB DECRYPTION

Likewise, if we have some ability to decrypt ECB cipher text, we can use the same block concept to decrypt other pieces of data.
Simply take the cipher text block in question and put it into the middle of data that you’re able to decrypt. So long as they use the same key, this data will decrypt cleanly and you’re able to compromise the cipher text in question.

MITIGATION

These flaws – and most others we’ll talk about – rely on fact: we may not be able to see the data or even guess at the contents, but we can tamper with the cipher text and the server will decrypt it.
The solution to this is simple: encrypt your data, then append a MAC of the encrypted data.

NEVER MAC-THEN-ENCRYPTION

You’ll note that I said before that you should encrypt first. This is critical, but something that many crypto protocols get wrong.
When you MAC then encrypt, you have to decrypt the data and then validate the MAC. This introduce a multitude of problems, such as padding oracles.

PADDING

Most commonly, data encrypted with a block cipher will not fall neatly on a block boundary and even if it does, there’s nothing stopping you from chopping blocks off the end; the data will be truncated, but will decrypt properly.

So we always pad data, even if it’s a multiple of the block size

PKCS#7

The most common padding system you’ll see in use is PKCS#7, and it’s extraordinarily simple.
If you need a byte of padding, it’s a single 01 byte.
If you need two bytes of padding, it’s two 02 bytes.
Etc

So if you have block size of 8, here are some samples:

daeken => daeken\x02\x02
hacker101 => hacker101\x07\x07\x07\x07\x07\x07\x07
somedatatogetherhere => somedatatogetherhere\x08\x08\x08\x08\x08\x08\x08\x08

Now padding has to be validated during decryption so that you can ensure that the data was received properly; if there’s a mismatch something went wrong.
In PKCS#7, you can simply look at the last byte of the last block and see how many padding bytes there are, then check that those all match.

PADDING ORACLE

Padding Oracle attacks come into play when you have CBC-mode data that is padded with PKCS#7.
If the server behaves differently when decrypting improperly padded data than properly padded data, this is an oracle – you can send it data and know whether or not it’s correctly padded.

If you remember, I mentioned that flipping a bit in one CBC cipher text block will cause that bit position to be flipped in the next block’s plain text.
Due to that simple bit of design, a padding oracle can allow us to use the server to completely decrypt data without knowing the key ourselves.

When exploiting a padding oracle, we start from the last byte of the second-to-last block.
Modifications there will affect the last byte of the last block of the data.
Our goals is to determine what byte of cipher text in block N-1 will cause the plaintext of block N to be 0x1 when XORed together. Once we know this, we know that the plain text of that byte in block N is cipher ^ 0x1.

So let’s say we have this data: hacker101\x07\x07\x07\x07\x07\x07\x07

We get out some cipher text blob; for the purposes of this explanation, we’ll say it’s:

deadbeefcafe0123
feedface456789ab

To decrypt the second block, we decrypt it using your cipher, the you XOR that with the first cipher text block to get your plaintext.

So with our example, we know that when you decrypt the block feedface456789ab, you must get something where the last byte ^ 0x23 (the last byte of the previous block) == 0x07. Any other value will cause a padding error – except one! If instead 0x07 our last byte became 0x01 that’s valid too; that means that there’s only one byte of padding.

So if we try all 255 other possible values for the last byte of our plain text, we know that one of them will certainly not give us a padding error, because it’ll set the byte to 0x01.
Once we have know this value, we simply XOR that byte with 0x03 (to change the final padding to 0x02) and repeat the same thing for the byte before it – one of them will make it so the last two bytes are 0x02 x02.

By performing this across all the bytes in a block, then walking back from there along the blocks, we can find the plain text value of every single byte of this data.
I cut out some details for simplicity, but this is the basic process. Google around for “padding oracle attack” and you’ll find more in-depth examples and code to execute the attack.

MITIGATION

As before, MACing your data after encryption – and validating the MAC before decryption – will solve this problem. However, many crypto protocols do not do this, and cannot be changed for compatibility reasons.
In this case, it’s imperative that you do not differentiate between errors. A user should never know why decryption failed; if it’s due to improper padding, that should look the same as having a bad digest or anything else.

HASH EXTENSION

With MD5, SHA1, and other Merkle-Damgard construction hashes, the digest that’s returned is really just the internal state of the hash algorithm.
This leads to one of the coolest practical crypto attacks.

To hash a given piece of data, the following steps are performed:

1. Initialize the hash state
2. For each block of data
   a) Mix it into the hash state
3. Pad the hash

4. Output the internal hash state

But what this means is that there’s nothing stopping you from simply continuing to add data to that output.
Take the digest, put it into the internal state of the algorithm, add new data, and pad it appropriately.

The actual extension of the hash is trivial, as we just saw. But what is our goal here?
If we have a block of data and the hash that corresponds to it. Let’s say it’s hash(‘secretkey’ + data)
With length extension, we can add to the end of data without knowing the secret key.

But because there’s padding between the old and new data (because it’s added by the algorithm), you need to figure out what that padding is. Then you simply add the hash padding into the data that goes along with it.
The actual hash padding will go at the end and be ignored as it always is, but we need to know how our data is being manipulated.

The process by which you can determine the padding is specific to the algorithm in question, but documented in many places.

MD5: http://www.skullsecurity.org/blog/2012/everything-you-need-to-know-about-hash-length-extension-attacks
SHA1: https://blog.whitehatsec.com/hash-length-extension-attacks

MITIGATION

If you notice, with an HMAC you had: hash(key + hash(key + message))
Due to the double-hashed nature of the structure, there’s no way you can length extend it.
When you are hashing a secret with data from the user, you should always use HMAC, for precisely this reason.