Back

Eli Sohl Cryptography Tutorial/Study Guide February 17, 2021 12 mins read

Cryptopals: Exploiting CBC Padding Oracles

This is a write-up of the classic padding oracle attack on CBC-mode block ciphers. If you've done the Cryptopals cryptography challenges, you'll remember it as challenge 17. This is a famous and elegant attack. With it, we will see how even a small data leak (in this case, the presence of a "padding oracle" – defined below) can lead to full plaintext recovery.

Like the Cryptopals challenges, this post is written to be accessible to anyone with an interest in cryptography – no graduate degree required. All you need is patience, focus, and some basic familiarity with the concepts in the following section.

Core Concepts

Here's what you'll need to know to follow the rest of this post. If any these definitions don't go into enough detail for you, please take a moment to look up their corresponding terms and do some more background reading before you continue.

Block cipher: AES is the most famous example. In general, a block cipher is a way of transforming (encrypting) fixed-size groups of bits so that they look random unless you possess the *key* used for encryption. With the key, you can recover (decrypt) the block's original contents. We'll be writing block encryption and decryption as ENC_{key} and DEC_{key} respectively. To encrypt messages of arbitrary size, a block cipher needs some extra scaffolding: a *block cipher mode* and a *padding scheme*.

Padding scheme: A way of "padding out" a message's length to the nearest multiple of the block length. The padding needs to be easily distinguishable from the message and easily removed after decryption. We're going to use a padding scheme known as PKCS#7, which works by appending n bytes of value n. This is not the only padding scheme for which the attack works, but it is the most common one.

Block cipher mode: A way of generalizing a block cipher to handle multi-block plaintexts. There are a lot of these; the one we'll be using here is called CBC mode (for Cipher Block Chaining).

CBC mode: A block cipher mode where each block of plaintext is XORed with the previous block's ciphertext prior to encryption (illustrated below). The first block of plaintext is XORed with a one-block *initialization vector*, which is commonly prepended to the ciphertext.

Support

Initialization vector: A one-block-sized bytestring with uniformly random contents. Commonly abbreviated to **IV**. This serves an important role: in CBC mode every n'th plaintext block is XORed against the (n-1)'th ciphertext block, but for the first plaintext block there is no previous ciphertext block to use, so this plaintext block is instead XORed against the IV. Among other benefits, the use of (distinct, randomly generated) IVs for each encryption ensures that multiple encryptions of the same plaintext will result in different, seemingly unrelated ciphertexts.

Padding oracle: Something which, given a ciphertext, tells us whether its decrypted plaintext has valid padding or not. The name is meant to evoke the similarly insightful and mysterious Oracles of antiquity. The classic example of a padding oracle is a service with detailed error messages. That said, the oracle could in fact be anything that allows us to differentiate between valid and invalid padding for arbitrary ciphertexts. Any number of side channels can lead to padding oracles. We'll look at a couple examples in the next section.

Sample Padding Oracles

OK, so as far as the attack is concerned, a padding oracle is a generic, abstract entity – but what might it look like in real life?

Say you're auditing a web API, and it authenticates its users using encrypted tokens. That is, it takes some data about the user (some of which might be secret from the user), encrypts this data under (say) AES-CBC using a key that is only stored server-side, and then it provides the user with the resulting ciphertext (which, you'll recall, looks just like a random string). You, the user, can't decrypt this ciphertext, but you can store it and provide it in future API queries. When you do this, the server will be able to decrypt it and instantly know all about you.

But what if you provide an invalid token?

There are two likely error states here. If the token's decryption has bad padding then the decryption operation will fail. On the other hand, if the plaintext has valid padding, the padding will be successfully stripped and processing will proceed to deserialization (which will almost certainly fail).

Suppose that the API decides to helpfully distinguish between these two error states, serving responses like `{'code': 401, 'msg': 'decryption error'}` or `{'code': 401, 'msg': 'deserialization error'}` respectively. This gives us a padding oracle! We can send any arbitrary ciphertext to the server and check the response's `msg` field. If we get 'decryption error', the padding is invalid; any other response indicates valid padding. As we are about to see, this is all we need to fully decrypt our – or anyone's – secret token.

Here's the second example. Suppose some altruistic cryptographer notices this issue we just described and reports it. The developers decide to fix the issue by changing their validation

code to return a generic error. If your token is invalid, they now just send back `{'code': 401}` with no context. No message, no oracle – or so it seems.

Support

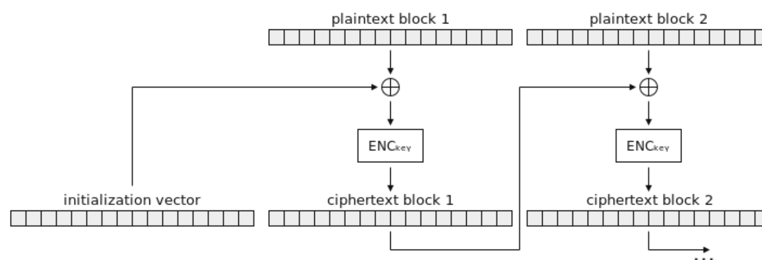
Well, it's true that we no longer have an error message to rely on, but we can still look at how long it takes the server to send their generic error. In this case, we might send each token a few times, watch how long it takes for the response to arrive, and infer that a longer delay indicates valid padding. If decryption and deserialization happen back-to-back, then the difference in timing will be very small (though still nonzero); if anything else takes place between these steps (e.g. writing logs, setting up a session object, opening a database connection, etc) then the timing difference will be that much easier to detect.

If we can establish reliably low-latency connections (e.g. if the server is hosted by a cloud provider and we spin up a virtual machine with that provider in the same region) then even a very small timing difference might be measurable. And if we are limited to less reliable connections, we can always just send more requests to get more timing data. We can consolidate this data to reach arbitrary levels of accuracy. This will slow down the attack and make it easier to detect, but the attack will still work.

Those are a couple examples of what a padding oracle might look like. Remember that as far as the rest of the attack is concerned, the details of the oracle aren't important; as long as we can take a ciphertext and determine, somehow, whether or not the ciphertext's decryption's padding is valid, that's all we need for this attack. In that sense, the oracle is effectively a black box.

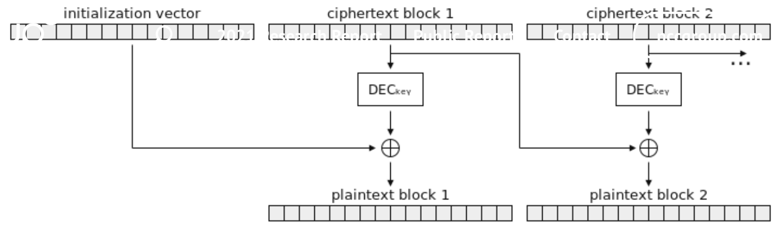
Enough preamble – let's look at how the attack actually works.

The Attack (Multi-Block Case)



This figure shows how multi-block messages are encrypted in CBC mode. Each plaintext block is XORed with the previous ciphertext block (or IV) prior to encryption. The arrangement of the bottom row of blocks here reflects how encrypted messages are usually serialized for transmission. While it is not strictly *required* for the IV to be prepended to the ciphertext — IV and ciphertext could be conveyed in some other way, e.g. as siblings in a JSON data structure — this form of serialization is something you will commonly see in practice.

We won't be spending much time with this multi-block figure, because it turns out that the multi-block case of this attack reduces nicely to the single-block case. To see why, let's take a quick look at the CBC decryption operation.

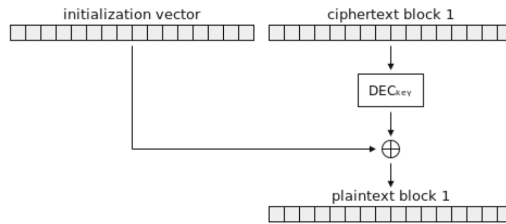


Support

As discussed above, without the encryption key we have no way of computing DEC_{key} directly. However, if we could somehow determine the output of DEC_{key}, the rest of the attack would just come down to bookkeeping. We could take the decrypted block, xor it against the previous ciphertext block (or IV), and thereby recover the corresponding block of plaintext. Do this for every single block, and we will have recovered the whole message. The only thing stopping us from doing this is the fact that we can't compute DEC_{key} directly.

However, as it turns out, recovering the output of DEC_{key} is precisely what the padding oracle attack allows us to do.

The Attack (Single-Block Case)



Here's that decryption operation again. This time, our encrypted message consists of an IV and a single ciphertext block.

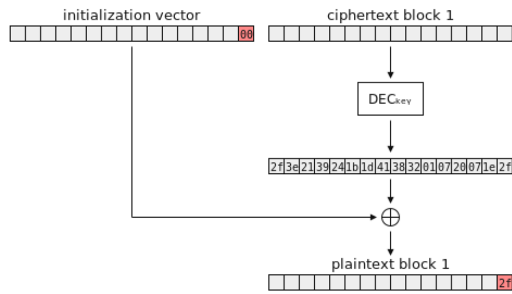
Say we pass an arbitrary block of ciphertext to our padding oracle. We can set the IV to whatever we want; we'll zero it for now. The illustration above shows what the oracle will compute. It doesn't tell us the result of this computation; it only tells us only whether or not the resulting plaintext block ends with valid padding.

The key idea behind the attack is this: by making modifications to the IV, we can predictably modify the plaintext block. Flipping a bit in the IV will flip the corresponding bit in the plaintext. Setting the IV's final byte to any value will xor that value into the plaintext's final byte. If we iterate through every possible value for the final IV byte, eventually one of them will set the plaintext's final byte to 0x01 – and our padding oracle will tell us when this happens, because 0x01 is valid padding!

Why is it valid? Recall that under the scheme we're using, valid padding consists of n bytes of value n. A trailing `0x01` byte might not look like much, but it meets this definition, so the oracle accepts it just like it would accept `0x02 0x02` or `0x03 0x03 0x03`.

Here's an example of what this looks like in action:

Support



You'll notice an extra block in this figure. This block shows the output of DEC_{key}. I've chosen totally arbitrary contents for this block; the point is just that you can see the relationship between this value, the IV, and the plaintext. In particular, the search for a valid IV byte ends when we reach `0x2e`, because `0x2e ⊕ 0x2f = 0x01`.

Once we have this step of the attack working, we can do something really cool: we can start to construct what I'll call a *zeroing IV*. This is an IV which will set some (eventually all) of the plaintext's bytes to zero.

Why zero? Two reasons, one of which is useful now and one of which will come up later. The reason I'll give you for now is this: zero gives us options. If we want to set a plaintext byte to any value other than zero, we can just xor that value into the zeroing IV. In other words, the zeroing IV gives us a way of manipulating the plaintext however we like.

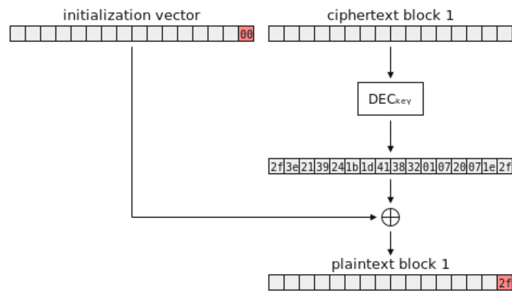
How do we build a zeroing IV? Well, as soon as we set the plaintext's final byte to `0x01`, we can take the corresponding IV byte and xor that against `0x01`. This modified IV byte will set the plaintext's final byte to `0x00` – and so it will work as the final byte of our zeroing IV.

Once we have that, we can derive a new IV which is guaranteed to set the plaintext's final byte to `0x02`, and we can start trying to set the plaintext's penultimate byte to `0x02` as well.

Actually, there is one tiny edge case that we have to check for first. Suppose the plaintext's penultimate byte is already set to 0x02. In this case, the message's padding would be valid if the final byte is set to either 0x01 or 0x02. If our search hits 0x02 before 0x01, but we assume that we found 0x01 and not 0x02, then the rest of the attack will fail. Luckily there is a simple test we can use here: as soon as we get an affirmative result from the oracle, we'll change the IV's penultimate byte and query the oracle again. If both queries succeed, this tells us that the penultimate byte is not part of the message's (valid) padding, proving that the padding has length one and thus must have value 0x01 as well. On the other hand, if this second query fails, we've run into a false positive and should keep searching.

Once we've found valid one-byte padding, we can use a similar process to search for valid two-byte padding. This search will go just like the search for the final byte (minus the edge case, since now we know our valid padding can only be of length 2). This search will terminate when the plaintext's final two bytes equal 0x02 0x02, at which point we'll know how to zero (and thus control) both of these bytes. This permits us to move on to attacking the third-from-last byte, then the fourth-from-last, and so on.

Here's what the full process looks like:



Support

This process can be followed until we've managed to build up a full zeroing IV. This brings us to the second reason why a zeroing IV is useful. One of the basic properties of xor is this: if `IV ⊕ BLOCK = 0`, then `IV = BLOCK`. In other words, we've just recovered the output of DEC_{key} – it is equal to our zeroing IV!

This is great! Now that we've recovered this, we can xor it against the previous ciphertext block (or IV) to recover the corresponding plaintext block. Run this process once per block and we'll have recovered the full plaintext!

Sample Implementation

That's the theory behind the attack. Here's how you might implement it.

Something to bear in mind: You can learn a lot from other people's code, but you'll learn even more from your own hands-on experience! My suggestion is that you go and try implementing the ideas above, then once you're done, come back to this reference implementation and use it to "check your work".

With that said, here's how I might write this attack:

```
#!/usr/bin/env python3

BLOCK_SIZE = 16


def single_block_attack(block, oracle):
    """Returns the decryption of the given ciphertext block"""

    # zeroing_iv starts out nulled. each iteration of the main loop will add
    # one byte to it, working from right to left, until it is fully populated,
    # at which point it contains the result of DEC(ct_block)
    zeroing_iv = [0] * BLOCK_SIZE

    for pad_val in range(1, BLOCK_SIZE+1):
        padding_iv = [pad_val ^ b for b in zeroing_iv]

        for candidate in range(256):
            padding_iv[-pad_val] = candidate
            iv = bytes(padding_iv)
            if oracle(iv, block):
                if pad_val == 1:
                    # make sure the padding really is of length 1 by changing
                    # the penultimate block and querying the oracle again
```

```
                    padding_iv[-2] ^= 1
                    iv = bytes(padding_iv)
                    if not oracle(iv, block):
                        continue  # false positive; keep searching
                break
        else:
            raise Exception("no valid padding byte found (is the oracle working
correctly?)")

        zeroing_iv[-pad_val] = candidate ^ pad_val

    return zeroing_iv


def full_attack(iv, ct, oracle):
    """Given the iv, ciphertext, and a padding oracle, finds and returns the plaintext"""
    assert len(iv) == BLOCK_SIZE and len(ct) % BLOCK_SIZE == 0

    msg = iv + ct
    blocks = [msg[i:i+BLOCK_SIZE] for i in range(0, len(msg), BLOCK_SIZE)]
    result = b''

    # loop over pairs of consecutive blocks performing CBC decryption on them
    iv = blocks[0]
    for ct in blocks[1:]:
        dec = single_block_attack(ct, oracle)
        pt = bytes(iv_byte ^ dec_byte for iv_byte, dec_byte in zip(iv, dec))
        result += pt
        iv = ct

    return result
```

This code provides a generic implementation of the attack. It will work for any reliable padding oracle. It demonstrates how the multi-block attack reduces to the single-block attack, how the single-block case builds up the zeroing IV one byte at a time, how to efficiently handle the edge case with the IV's first byte, and so on.

To attack specific padding oracles, one can just import this script and set it to work. For example, here's a little script that imports the previous code snippet and uses it to solve [Cryptopals Challenge 17](#).

Note that to save on space, I've chosen to use the versions of AES, CBC, and PKCS#7 provided by the `PyCryptodome` library rather than including full implementations of those building blocks here.

```
#!/usr/bin/env python3

import random
import os

from Crypto.Cipher import AES  # requires PyCryptodome
from Crypto.Util.Padding import pad, unpad
```

```
class Challenge:
    _strings = (
        b"MDAwMDAwTm93IHRoYXQgdGhlIHBhcnR5IGlzIGp1bXBpbmc=",
        b"MDAwMDAxV2l0aCB0aGUgYmFzcyBraWNrZWQgaW4gYW5kIHRoZSBWZWdhJ3MgYXJlIHB1bXBpbic=",
        b"MDAwMDAyUXVpY2sgdG8gdGhlIHBvaW50LCB0byB0aGUgcG9pbnQsIG5vIGZha2luZw==",
        b"MDAwMDAzQ29va2luZyBNQydzIGxpa2UgYSBwb3VuZCBvZiBiYWNvbg==",
        b"MDAwMDA0QnVybmluZyAnZW0sIGlmIHlvdSBhaW4ndCBxdWljayBhbmQgbmltYmxl",
        b"MDAwMDA1SSBnbyBjcmF6eSB3aGVuIEkgaGVhciBhIGN5bWJhbA==",
        b"MDAwMDA2QW5kIGEgaGlnaCBoYXQgd2l0aCBhIHNvdXBlZCB1cCB0ZW1wbw==",
        b"MDAwMDA3SSdtIG9uIGEgcm9sbCwgaXQncyB0aW1lIHRvIGdvIHNvbG8=",
        b"MDAwMDA4b2xsaW4nIGluIG15IGZpdmUgcG9pbnQgb2g=",
        b"MDAwMDA5aXRoIG15IHJhZy10b3AgZG93biBzbyBteSBoYWlyIGNhbiBibG93"
    )

    def __init__(self):
        self._key = os.urandom(16)

    def get_string(self):
        """This is the first function described by Challenge 17."""
        string = random.choice(self._strings)
        cipher = AES.new(self._key, AES.MODE_CBC)
        ct = cipher.encrypt(pad(string, AES.block_size))
        return cipher.iv, ct

    def check_padding(self, iv, ct):
        """This is the second function described by Challenge 17."""
        cipher = AES.new(self._key, AES.MODE_CBC, iv)
        pt = cipher.decrypt(ct)
        try:
            unpad(pt, AES.block_size)
        except ValueError:  # raised by unpad() if padding is invalid
            return False
        return True


if __name__ == "__main__":
    from cbc_padding_oracle_attack import full_attack
    from base64 import b64decode

    service = Challenge()
    iv, ct = service.get_string()
    print("Ciphertext:", ct)
    print("Launching attack...")

    result = full_attack(iv, ct, service.check_padding)
    plaintext = unpad(result, AES.block_size)
    print("Recovered plaintext:", plaintext)
    print("Decoded:", b64decode(plaintext).decode('ascii'))
```

Defenses

Support

This attack is a chosen-ciphertext attack. It depends on the attacker being able to submit arbitrary ciphertexts to the oracle. As such, you can prevent the attack by *authenticating your ciphertexts*. You might do this by switching from CBC mode to an authenticated encryption mode like GCM or OCB; alternately, keep CBC mode but start MACing your ciphertexts using something like HMAC.

Removing the oracle would also prevent the attack. However, hopefully the example oracles above gave you some sense of how nontrivial this actually can be in practice. This is a cryptographic problem and it calls for a cryptographic solution; anything less is likely to be fragile and error-prone.

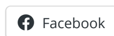
By adding authentication tags and checking them prior to decryption, we guarantee that we'll be able to reject any attacker-crafted messages without ever decrypting them, preventing us from leaking any information at all about their decrypted contents, padding-related or otherwise.

Share this:

Twitter

Reddit

LinkedIn



Facebook

Like this:

Loading...

Published by Eli Sohl

View all posts by Eli Sohl ->

Here are some related articles you may find interesting

Technical Advisory: Adobe ColdFusion WDDX Deserialization Gadgets

Multiple vulnerabilities identified in Adobe ColdFusion allow an unauthenticated attacker to obtain the service account NTLM password hash, verify the existenc...

- Technical advisories
- Vulnerability Research

November 21, 2023

15 mins read

Is this the real life? Is this just fantasy? Caught in a landslide, NoEscape from NCC Group

Author: Alex Jessop (@ThisIsFineChief) Summary Tl;dr This post will delve into a recent incident response engagement handled by NCC Group's Cyber...

- Detection and Threat Hunting
- Digital Forensics and Incident Response (DFIR)
- Threat Intelligence

November 20, 2023

7 mins read

The Spelling Police: Searching for Malicious HTTP Servers by Identifying Typos in HTTP Responses

At Fox-IT (part of NCC Group) identifying servers that host nefarious activities is a critical aspect of our threat intelligence. One approach involves looking fo...

- Cyber Security
- Detection and Threat Hunting
- Fox-IT
- Fox-IT and European Research
- Research

November 15, 2023

7 mins read

Previous post

Next post

View articles by category

- 5G Security & Smart Environments (10)
- Academic Partnership (3)
- Annual Research Report (2)
- Asia Pacific Research (1)
- Awards & Recognition (4)

Most popular posts

Most recent posts

- Technical Advisory: Adobe ColdFusion WDDX Deserialization Gadgets
- Is this the real life? Is this just fantasy? Caught in a landslide, NoEscape from NCC Group
- The Spelling Police: Searching for Malicious HTTP Servers by Identifying Typos in HTTP Responses
- Public Report – WhatsApp Auditable Key Directory (AKD) Implementation Review

- Blockchain (3)
- Books (17)
- Business Insights (6)
- Cloud & Containerization (34)
- Cloud Security (18)
- Conferences (37)
- Corporate (7)
- Cryptography (112)
- CTFs/Microcorruption (1)
- Current events (1)
- Cyber as a Science (6)
- Cyber Security (402)
- Detection and Threat Hunting (16)
- Digital Forensics and Incident Response (DFIR) (20)
- Disclosure Policy (1)
- Emerging Technologies (11)
- Engineering (5)
- Fox-IT (16)
- Fox-IT and European Research (6)
- Gaming & Media (8)
- Hardware & Embedded Systems (103)
- Intern Projects (2)
- iSec Partners (52)
- Machine Learning (28)
- Managed Detection & Response (22)
- Misinformation, Deepfakes, & Synthetic Media (2)

Don't throw a hissy fit; defend against Medusa

Support

- North American Research (28)
- Offensive Security & Artificial Intelligence (13)
- Patch notifications (35)
- Presentations (55)
- protocol_name (1)
- Public interest technology (10)
- Public interest technology (1)
- Public Reports (43)
- Public tools (105)
- Reducing Vulnerabilities at Scale (22)
- Research (362)
- Research Paper (20)
- Resources (1)
- Reverse Engineering (47)
- Risk Management & Governance (6)
- Standards (13)
- Technical advisories (215)
- Technology Policy (1)
- Threat briefs (3)
- Threat Intelligence (67)
- Tool Release (106)
- Transport (16)
- Tutorial/Study Guide (46)
- UK Research (9)
- Uncategorized (25)

Support

- Virtualization, Emulation, & Containerization (10)
- VSR (32)
- Vulnerability (164)
- Vulnerability Research (4)
- Whitepapers (238)

Support

Call us before you need us.

Our experts will help you.

Get in touch

Call us on:

General Number:

441612095200

24/7 Emergency Incident Response:

443316300690

© NCC Group 2023. All rights reserved.