Exploiting padding oracles with fixed IVs

Exploiting padding oracles with fixed IVs - Author not stated, Teddy Katz's Blog.

- Published: 2019-11-23
- Original: <https://blog.teddykatz.com/2019/11/23/json-padding-oracles.html>
- Preserved from: https://blog.teddykatz.com/2019/11/23/json-padding-oracles.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

A few weeks ago, I found an interesting account takeover bug in a webserver. This is the type of issue where it's not too difficult to spot that something is wrong, but it's surprisingly complicated to create a working exploit, requiring several different attacks to be chained together. In the process, I also ended up learning a lot about modern cryptography algorithms.

The bug bounty program that maintains the project has broadly prohibited public writeups of security reports, so I've tried to anonymize the bug by creating a contrived capture-the-flag problem that has the same vulnerability.

(To run this yourself, download `server.js` above, put files called `key.txt` and `flag.txt` in the same directory containing anything you want, and run `node server.js`.)

To summarize: The server generates encrypted per-user tokens, which decrypt to JSON objects like `{ time: <timestamp>, username: <username> }`. Anyone can get a token for the `anonymous` account by navigating to `/`. The flag can be obtained at `/flag?token=<TOKEN>`, where `<TOKEN>` is a token for the `admin` account.

The challenge is to obtain the contents of `flag.txt` as a client of the server, without having access to the filesystem. If you'd like to try the challenge yourself without hints, stop reading here.

Encryption without signatures

To get the flag, we need to somehow create a valid token for the `admin` account.

The first thing to notice is that tokens are encrypted, but not signed. This means that if we receive a token, we can arbitrarily change parts of it to cause it to decrypt to something else.

For example, when I visit `/` my token is a long hex string starting with `dbd53ef0...`. Unsurprisingly, if I try to use that token to get the flag by visiting `/flag?token=dbd53ef0...`, I get a "wrong user" error,

because the token was generated for the `anonymous` account rather than the `admin` account. But if I change the first character from an `d` to an `e`, I get a different error: `Unexpected token Ã« in JSON at position 0`. In other words, we were able to create an auth token that decrypts to something else (which is apparently invalid JSON), just by arbitrarily changing a character.

A valid token for the `admin` account is a token that decrypts to a JSON object like `{ time: <timestamp>, username: "admin" }`. So we need to somehow create a token that decrypts to a string that we choose, rather than just decrypting to something random. To figure out how we can do this, it's helpful to understand how the encryption process works.

CBC overview

The tokens are generated using the `AES-192-CBC` algorithm. What does that mean?

-

AES-192 is a type of encryption protocol known as a *block cipher*. Given some secret key, AES-192 takes a 16-byte *block* as input, and produces a different 16-byte block as output. The details of how it does this aren't really relevant here – we can just treat it as a black box that takes a 16-byte string *x* and spits out an encrypted 16-byte string *E(x)*.

The process is also reversible; given the secret key and an encrypted 16-byte string *E(x)*, we can use the decryption function *D* to compute *D(E(x))*, which is equal to *x*. (In other words, if you encrypt a 16-byte string and then decrypt it, you'll get the original string back again.)

-

What if we want to encrypt a message that isn't exactly 16 bytes long? To achieve this, we can split the message into 16-byte chunks. We could just encrypt each chunk individually and join together the encrypted blocks, but this isn't a good idea, for reasons that are outside the scope of this post). Instead, some modern protocols use a process called Cipher Block Chaining) (CBC).

CBC works as follows:

-

Decide on some message to encrypt (plaintext).

`hi, how have you been, how was your day today?`

-

Split the plaintext into 16-byte blocks. Call the blocks *P1*, *P2*, *P3*, etc.

- *P1*: `hi, how have you`
- *P2*: `been, how was y`
- *P3*: `our day today?`

-

To encrypt the first plaintext block, *P1*, first bitwise XOR it with a 16-byte string called the *Initialization Vector* (IV, also known as *C0*), and encrypt the result with AES to obtain a 16-byte ciphertext block *C1*. (More on the choosing the IV later.)

C1 = E(P1 ^ IV)

-

To encrypt the second plaintext block, first XOR it with *C1*, and then encrypt the result to obtain the ciphertext block *C2*. To encrypt the third plaintext block, first XOR it with *C2*, then encrypt the result to obtain *C3*. Repeat the process (XORing the plaintext block with the previous ciphertext block, and encrypt it to obtain the new ciphertext block) until all the plaintext blocks are encrypted.

C2 = E(P2 ^ C1)

C3 = E(P3 ^ C2)

...

-

The final encrypted message is just the result of joining all of the ciphertext blocks together.

The name "Cipher Block Chaining" comes from the fact that each ciphertext block depends on the block before it, so computing all of the ciphertext blocks requires "chaining" them together.

To decrypt the message, just do the inverse of each step. So:

- *P3 = D(C3) ^ C2*
- *P2 = D(C2) ^ C1*
- *P1 = D(C1) ^ IV*

Padding oracles

As a reminder, our goal is to somehow create a token that will decrypt to a chosen plaintext. We don't know the encryption key, so we can't directly encrypt or decrypt anything (otherwise, this would be easy: we could achieve our goal by just encrypting our chosen plaintext). However, we can do something similar using a clever technique known as a padding oracle attack.

A padding oracle lets us do the following: Given a desired plaintext block *Pn* and any arbitrary ciphertext block *Cn*, we can compute a block *Cn-1* such that *D(Cn) ^ Cn-1 = Pn*.

(I won't go into detail here on how *Cn-1* is computed, because padding oracle attacks are very well-known and this post is getting long. If you're interested, here is a good explanation of how padding oracles work.)

This seems promising. Let's choose a plaintext of `{"time":1574477443310,"user":"admin"}` (a token for the admin account), and split it into chunks:

- *P1*: `{"time":15744774`
- *P2*: `43310,"user":"ad`
- *P3*: `min"}`

Then we can start generating ciphertext blocks:

- Set *C3* to a random 16-byte string.
- Since we know *C3* and *P3*, we can use the padding oracle to generate *C2* such that *D(C3) ^ C2 = P3*.

- Now we know *C2* and *P2*, so we can use the padding oracle to generate *C1* such that *D(C2)* ^ *C1* = *P2*.
- Finally, we know *C1* and *P1*, so we can use the padding oracle to generate an IV (*C0*) such that *D(C1)* ^ *C0* = *P1*.

And it seems like we're done – we've generated ciphertext blocks *C0*, *C1*, *C2*, and *C3* that will decrypt to our plaintext blocks *P1*, *P2*, and *P3*. But there's a bit of a problem – if our ciphertext is just composed of *C1*, *C2*, and *C3*, how do we tell the server to use *C0* as the IV?

The fixed-IV problem

As a reminder, the Initialization Vector (IV) is a 16-byte string used in the encryption and decryption process. Anyone who's supposed to decrypt a ciphertext needs to know the IV somehow. There are several ways to generate an IV to make sure the receiver knows it:

- Generate the IV at random for every message, and include it along with the ciphertext (so the transmitted ciphertext is *C0*, *C1*, *C2*, ...).
- Use a fixed IV for every message (say, by deterministically generating it from the secret key). Then the transmitted ciphertext is *C1*, *C2*, ... and the receiver knows *C0* out-of-band. (Note: This is generally not a good idea – if you need an IV and you're not sure what to choose, go with (1).)

In our case, we've used a padding oracle to generate blocks *C0*, *C1*, *C2*, and *C3* such that if the server uses *C0* as the IV to decrypt the ciphertext *C1*, *C2*, and *C3*, then we will get our chosen plaintext. So if the server used strategy (1), then we would be completely done – we would just tell the server that the IV was *C0*, and the server would use it.

Unfortunately for us, it turns out that Node's `crypto.createCipher` function uses strategy (2). (Again, this is generally not a good idea, and in fact `crypto.createCipher` is deprecated in favor of a method that uses strategy (1).) This poses a major problem here, because we can't tell the server what IV to use; it's just going to use a fixed IV that's deterministically generated from the key.

JSON parsers vs. random junk

Where does that leave us? Let's pretend the problem from the last section doesn't exist, and just send the server a ciphertext of *C0*, *C1*, *C2*, and *C3* to see what happens when the server tries to decrypt it using its own IV.

Since we're sending four ciphertext blocks, the server will decrypt our ciphertext to four different plaintext blocks.

- To decrypt the last block, the server will compute *D(C3)* ^ *C2* to get a result of *P3*.
- To decrypt the second-to-last block, the server will compute *D(C2)* ^ *C1* to get a result of *P2*.
- To decrypt the third-to-last block, the server will compute *D(C1)* ^ *C0* to get a result of *P1*.

Seems fine so far – the last three plaintext blocks are our desired plaintext.

- To decrypt the fourth-to-last (i.e. first) block, the server will compute *D(C0)* ^ *IV* to get sixteen bytes of random junk.

So the server will decrypt our token to `RANDOM_JUNK_HERE{"time":1574477443310,"user":"admin"}` (where `RANDOM_JUNK_HERE` is a placeholder for sixteen pseudorandom bytes).

This is a problem, because after the server decrypts our token, it needs to parse the results as JSON. And prepending sixteen bytes of uncontrollable random junk to the start of a JSON string is a pretty surefire way to create invalid JSON, which is going to cause the server to return a 500 error instead of giving us the flag.

(By the way, it wouldn't have been any better if we'd omitted *C0* and just sent *C1*, *C2*, and *C3* to the server. The server would have then decrypted the third-to-last block as *D(C1)* ^ *IV* to get a result of *P1* ^ *C0* ^ *IV*, which would again effectively be random junk. So the final decrypted token would be `RANDOM_JUNK_HERE43310,"user":"admin"}` which is still unlikely to be valid JSON.)

Moving random junk

At this point, I was stumped for awhile. I had figured out how to create a token that would decrypt to something like `RANDOM_JUNK_HERE{"time":1574477443310,"user":"admin"}`, but I couldn't figure out how to get rid of the random junk so that the plaintext would parse as JSON.

After thinking about it for a few hours, I had an idea: Instead of trying to get rid of the random junk, what if we could *move* it to the middle of the JSON object? Maybe it would be more likely to parse there.

How do we move the random junk?

I made a few observations:

- The server is happy to provide us with tokens for the `anonymous` account, which decrypt to a JSON object like `{"time":1574477443310,"user":"anonymous"}`.
- A decrypted token for the `anonymous` account starts out the same way as a decrypted token for the `admin` account. (The usernames themselves are near the end of the plaintext string.)

The server is encrypting tokens using a fixed IV. What will happen if it encrypts two different plaintexts that start with the same sixteen bytes?

Well, the first ciphertext block only depends on the first plaintext block (and the IV, which is fixed). So the first block of ciphertext will be identical for both plaintexts.

Similarly, the second ciphertext block only depends on the first two plaintext blocks. So if two plaintexts start with the same 32 bytes, then the first two blocks of ciphertext will be identical for both plaintexts. This is true for any multiple of sixteen bytes: If two plaintexts both share a `16c`-byte prefix, then their ciphertexts will also share a `16c`-byte prefix. (By the way, this is why it's generally not a good idea to use a fixed IV.)

So if we want to have ciphertext blocks at the start of a token that decrypt to something other than random junk, then it's very easy: we can just reuse the blocks from the `anonymous` token that the server gives us for free.

Let's suppose we take the first sixteen bytes of the `anonymous` token from the server, and call the resulting block *C'*. Then we run the padding oracle attack with a chosen plaintext of `43310,"user":"admin"}` (i.e. the desired admin JSON object, with the first sixteen bytes removed) to

obtain additional ciphertext blocks *C0*, *C1*, and *C2*. What will the server do if we send the ciphertext containing *C'*, *C0*, *C1*, and *C2*?

- To decrypt the first block, the server will compute *D(C')* ^ *IV*. This must be equal to the plaintext `{"time":15744774` (i.e. the first block of the full anonymous token), because that exact same computation would be used when decrypting the anonymous token.
- To decrypt the second block, the server will compute *D(C0)* ^ *C'* to get sixteen bytes of random junk.
- To decrypt the third block, the server will compute *D(C1)* ^ *C0* to get a result of `43310,"user":"ad`, the first sixteen bytes of the chosen plaintext.
- To decrypt the fourth block, the server will compute *D(C2)* ^ *C1* to get a result of `min"}`, the rest of the chosen plaintext.

The resulting plaintext is something like `{"time":15744774RANDOM_JUNK_HERE43310,"user":"admin"}`. We've successfully moved the random junk to the middle of the plaintext!

That's nice, but how was that helpful?

JSON parsers vs. well-positioned random junk

Using the strategy described in the last section, we can now move the random junk into the middle of the plaintext, provided that:

- The random junk is at a multiple-of-sixteen-byte offset in the plaintext.
- Everything that precedes the random junk in the plaintext is also present in the anonymous JSON object that we receive from the server.

(Note: there are no restrictions on what comes *after* the random junk – we can put anything we want there.)

As a reminder, we need this plaintext-with-random-junk thing to parse as valid JSON, otherwise the server will throw an error before it even has a chance to give us the flag. But in the last example, we ended up with a plaintext like `{"time":15744774RANDOM_JUNK_HERE43310,"user":"admin"}`. Since the random junk appears in the middle of a JSON number, it would have to consist entirely of ASCII digits (or a specific combination of quotes and commas) for the result to be valid JSON. This is pretty unlikely.

What if we could put the random junk into a JSON *string*? JSON strings can tolerate a much larger variety of characters, so we might have a better chance of parsing successfully. Conveniently, if we put the junk at a 32-byte offset instead of a 16-byte offset, it ends up right in the middle of a JSON string. The result looks like `{"time":1574477443310,"user":"anRANDOM_JUNK_HEREmin"}`.

That seems better, but will random junk be valid JSON even within a string? Of the 256 possible Latin-1 characters, all of them are valid in JSON strings except the 32 control characters, double quotes (which will cause problems by ending the string), and backslashes (which will sometimes cause problems by creating invalid escapes). That leaves 222 "safe" characters that won't cause parsing errors. The chance that every character in our 16-byte random junk block will be safe is (222 / 256)16 ≈ 0.102. So there's about a 10% chance that we'll end up with valid JSON.

What should we do if we're unlucky? Well, the junk is effectively a complicated function of the server's secret key, the desired plaintext, and... the random 16-byte string that we chose as the rightmost ciphertext block when starting the padding oracle attack.

So if the random junk ends up making the JSON invalid, we can just reroll it by choosing a different random string and redoing the whole padding oracle exploit. After 10 tries or so, we'll have valid JSON.

Repairing payload damage caused by random junk

We're almost there! After the last step, we've figured out how to create a token that decrypts to something like `{"time":1574477443310,"user":"anRANDOM_JUNK_HEREmin"}`, and we can make sure the plaintext is valid JSON too.

There's one problem left: the random junk is kind of in the way. In order to make the random junk appear within a string, we had to put it inside the `user` property (since it could only be placed at a multiple-of-sixteen-byte offset in the plaintext). But when the server parses the JSON object, we need the `user` property to be the string `"admin"` if it's going to give us the flag. `"anRANDOM_JUNK_HEREmin"` isn't going to cut it.

I was stumped again for awhile here. Eventually, I had another idea: If we don't like the contents of the `user` property, why don't we just create *another* `user` property? We can put anything we want after the random junk, including another JSON property. And conveniently, when the JS `JSON.parse` function encounters multiple properties with the same name, it keeps the *last* one.

So we can:

- Take the first 32 bytes of an `anonymous` token from the server,
- ...use a padding oracle with a payload of `","user":"admin"}` to get a set of ciphertext blocks,
- ...concatenate everything,
- ...repeat 10 times or so until the result decrypts to valid JSON,

-

...and end up with a token that decrypts to a JSON plaintext of

`{"time":1574477443310,"user":"anRANDOM_JUNK_HERE","user":"admin"}`.

And that's how we can escalate to admin privileges and get the flag.

Exploit code (prints out the flag after a couple minutes):
