Security

After reading this guide, you'll know:

1. The security surface area of a Meteor app.
2. How to secure Meteor Methods, publications, and source code.
3. Where to store secret keys in development and production.
4. How to follow a security checklist when auditing your app.
5. How App Protection works in Galaxy Hosting.

Introduction

Securing a web application is all about understanding security domains and understanding the attack surface between these domains. In a Meteor app, things are pretty simple:

1. Code that runs on the server can be trusted.
2. Everything else: code that runs on the client, data sent through Method and publication arguments, etc, can't be trusted.

In practice, this means that you should do most of your security and validation on the boundary between these two domains. In simple terms:

1. Validate and check all inputs that come from the client.
2. Don't leak any secret information to the client.

Concept: Attack surface

Since Meteor apps are often written in a style that puts client and server code together, it's extra important to be aware what is running on the client, what is running on the server, and what the boundaries are. Here's a complete list of places security checks need to be done in a Meteor app:

1. **Methods**: Any data that comes in through Method arguments needs to be validated, and Methods should not return data the user shouldn't have access to.
2. **Publications**: Any data that comes in through publication arguments needs to be validated, and publications should not return data the user shouldn't have access to.
3. **Served files**: You should make sure none of the source code or configuration files served to the client have secret data.

Each of these points will have their own section below.

Avoid allow/deny

In this guide, we're going to take a strong position that using allow or deny to run MongoDB queries directly from the client is not a good idea. The main reason is that it is hard to follow the principles outlined above. It's extremely difficult to validate the complete space of possible MongoDB operators, which could potentially grow over time with new versions of MongoDB.

There have been several articles about the potential pitfalls of accepting MongoDB update operators from the client, in particular the Allow & Deny Security Challenge and its results, both on the Discover Meteor blog.

Given the points above, we recommend that all Meteor apps should use Methods to accept data input from the client, and restrict the arguments accepted by each Method as tightly as possible.

Here's a code snippet to add to your server code which disables client-side updates on a collection. This will make sure no other part of your app can use `allow`:

```
// Deny all client-side updates on the Lists collection
Lists.deny({
  insert() { return true; },
  update() { return true; },
  remove() { return true; },
});
```

Methods

Methods are the way your Meteor server accepts inputs and data from the outside world, so it's natural that they are the most important topic for security. If you don't properly secure your Methods, users can end up modifying your database in unexpected ways - editing other people's documents, deleting data, or messing up your database schema causing the app to crash.

Validate all arguments

It's much easier to write clean code if you can assume your inputs are correct, so it's valuable to validate all Method arguments before running any actual business logic. You don't want someone to pass a data type you aren't expecting and cause unexpected behavior.

Consider that if you are writing unit tests for your Methods, you would need to test all possible kinds of input to the Method; validating the arguments restricts the space of inputs you need to unit test, reducing the amount of code you need to write overall. It also has the extra bonus of being self-documenting; someone else can come along and read the code to find out what kinds of parameters a Method is looking for.

Just as an example, here's a situation where not checking arguments can be disastrous:

```
Meteor.methods({
  removeWidget(id) {
    if (! this.userId) {
      throw new Meteor.Error('removeWidget.unauthorized');
    }

    Widgets.remove(id);
  }
});
```

If someone comes along and passes a non-ID selector like `{}`, they will end up deleting the entire collection.

mdg:validated-method

To help you write good Methods that exhaustively validate their arguments, we've written a wrapper package for Methods that enforces argument validation. Read more about how to use it in the Methods article. The rest of the code samples in this article will assume that you are using this package. If you aren't, you can still apply the same principles but the code will look a little different.

Don't pass userId from the client

The `this` context inside every Meteor Method has some useful information about the current connection, and the most useful is `this.userId`. This property is managed by the DDP login system, and is guaranteed by the framework itself to be secure following widely-used best practices.

Given that the user ID of the current user is available through this context, you should never pass the ID of the current user as an argument to a Method. This would allow any client of your app to pass any user ID they want. Let's look at an example:

```
// #1: Bad! The client could pass any user ID and set someone else's name
setName({ userId, newName }) {
  Meteor.users.update(userId, {
    $set: { name: newName }
  });
}

// #2: Good, the client can only set the name on the currently logged in user
setName({ newName }) {
  Meteor.users.update(this.userId, {
    $set: { name: newName }
  });
}
```

The *only* times you should be passing any user ID as an argument are the following:

1. This is a Method only accessible by admin users, who are allowed to edit other users. See the section about user roles to learn how to check that a user is in a certain role.
2. This Method doesn't modify the other user, but uses it as a target; for example, it could be a Method for sending a private message, or adding a user as a friend.

One Method per action

The best way to make your app secure is to understand all of the possible inputs that could come from an untrusted source, and make sure that they are all handled correctly. The easiest way to understand what inputs can come from the client is to restrict them to as small of a space as possible. This means your Methods should all be specific actions, and shouldn't take a multitude of options that change the behavior in significant ways. The end goal is that you can look at each Method in your app and validate or test that it is secure. Here's a secure example Method from the Todos example app:

```
export const makePrivate = new ValidatedMethod({
  name: 'lists.makePrivate',
  validate: new SimpleSchema({
    listId: { type: String }
  }).validator(),
  run({ listId }) {
    if (!this.userId) {
      throw new Meteor.Error('lists.makePrivate.notLoggedIn',
        'Must be logged in to make private lists.');
    }

    const list = Lists.findOne(listId);

    if (list.isLastPublicList()) {
      throw new Meteor.Error('lists.makePrivate.lastPublicList',
        'Cannot make the last public list private.');
    }

    Lists.update(listId, {
      $set: { userId: this.userId }
    });

    Lists.userIdDenormalizer.set(listId, this.userId);
  }
});
```

You can see that this Method does a *very specific thing* - it makes a single list private. An alternative would have been to have a Method called `setPrivacy`, which could set the list to private or public, but it turns out that in this particular app the security considerations for the two related operations - `makePrivate` and `makePublic` - are very different. By splitting our operations into different Methods, we make each one much clearer. It's obvious from the above Method definition which arguments we accept, what security checks we perform, and what operations we do on the database.

However, this doesn't mean you can't have any flexibility in your Methods. Let's look at an example:

```
Meteor.users.methods.setUserData = new ValidatedMethod({
  name: 'Meteor.users.methods.setUserData',
  validate: new SimpleSchema({
    fullName: { type: String, optional: true },
    dateOfBirth: { type: Date, optional: true },
  }).validator(),
  run(fieldsToSet) {
    Meteor.users.update(this.userId, {
      $set: fieldsToSet
    });
  }
});
```

The above Method is great because you can have the flexibility of having some optional fields and only passing the ones you want to change. In particular, what makes it possible for this Method is that the security considerations of setting one's full name and date of birth are the same - we don't have to do different security checks for different fields being set. Note that it's very important that the `$set` query on MongoDB is generated on the server - we should never take MongoDB operators as-is from the client, since they are hard to validate and could result in unexpected side effects.

Refactoring to reuse security rules

You might run into a situation where many Methods in your app have the same security checks. This can be simplified by factoring out the security into a separate module, wrapping the Method body, or extending the `Mongo.Collection` class to do security inside the `insert`, `update`, and `remove` implementations on the server. However, implementing your client-server communication via specific Methods is still a good idea rather than sending arbitrary `update` operators from the client, since a malicious client can't send an `update` operator that you didn't test for.

Rate limiting

Like REST endpoints, Meteor Methods can be called from anywhere - a malicious program, script in the browser console, etc. It is easy to fire many Method calls in a very short amount of time. This means it can be easy for an attacker to test lots of different inputs to find one that works. Meteor has built-in rate limiting for password login to stop password brute-forcing, but it's up to you to define rate limits for your other Methods.

In the Todos example app, we use the following code to set a basic rate limit on all Methods:

```
// Get list of all method names on Lists
const LISTS_METHODS = _.pluck([
  insert,
  makePublic,
  makePrivate,
  updateName,
  remove,
], 'name');

// Only allow 5 list operations per connection per second

if (Meteor.isServer) {
  DDPRateLimiter.addRule({
    name(name) {
      return _.contains(LISTS_METHODS, name);
    },

    // Rate limit per connection ID
    connectionId() { return true; }
  }, 5, 1000);
}
```

This will make every Method only callable 5 times per second per connection. This is a rate limit that shouldn't be noticeable by the user at all, but will prevent a malicious script from totally flooding the server with requests. You will need to tune the limit parameters to match your app's needs.

If you're using validated methods, there's an available ddp-rate-limiter-mixin.

Publications

Publications are the primary way a Meteor server can make data available to a client. While with Methods the primary concern was making sure users can't modify the database in unexpected ways, with publications the main issue is filtering the data being returned so that a malicious user can't get access to data they aren't supposed to see.

You can't do security at the rendering layer In a server-side-rendered framework like Ruby on Rails, it's sufficient to not display sensitive data in the returned HTML response. In Meteor, since the rendering is done on the client, an `if` statement in your HTML template is not secure; you need to do security at the data level to make sure that data is never sent in the first place.

Rules about Methods still apply

All of the points above about Methods apply to publications as well:

1. Validate all arguments using `check` or npm `simpl-schema`.
2. Never pass the current user ID as an argument.
3. Don't take generic arguments; make sure you know exactly what your publication is getting from the client.
4. Use rate limiting to stop people from spamming you with subscriptions.

Always restrict fields

`Mongo.Collection#find` has an option called `fields` which lets you filter the fields on the fetched documents. You should always use this in publications to make sure you don't accidentally publish secret fields.

For example, you could write a publication, then later add a secret field to the published collection. Now, the publication would be sending that secret to the client. If you filter the fields on every publication when you first write it, then adding another field won't automatically publish it.

```
// #1: Bad! If we add a secret field to Lists later, the client
// will see it
Meteor.publish('lists.public', function () {
  return Lists.find({userId: {$exists: false}});
});

// #2: Good, if we add a secret field to Lists later, the client
// will only publish it if we add it to the list of fields
Meteor.publish('lists.public', function () {
  return Lists.find({userId: {$exists: false}}, {
    fields: {
      name: 1,
      incompleteCount: 1,
      userId: 1
    }
  });
});
```

If you find yourself repeating the fields often, it makes sense to factor out a dictionary of public fields that you can always filter by, like so:

```
// In the file where Lists is defined
Lists.publicFields = {
  name: 1,
  incompleteCount: 1,
  userId: 1
};
```

Now your code becomes a bit simpler:

```
Meteor.publish('lists.public', function () {
  return Lists.find({userId: {$exists: false}}, {
    fields: Lists.publicFields
  });
});
```

Publications and userId

The data returned from publications will often be dependent on the currently logged in user, and perhaps some properties about that user - whether they are an admin, whether they own a certain document, etc.

Publications are not reactive, and they only re-run when the currently logged in `userId` changes, which can be accessed through `this.userId`. Because of this, it's easy to accidentally write a publication that is secure when it first runs, but doesn't respond to changes in the app environment. Let's look at an example:

```
// #1: Bad! If the owner of the list changes, the old owner will still see it
Meteor.publish('list', function (listId) {
  check(listId, String);

  const list = Lists.findOne(listId);

  if (list.userId !== this.userId) {
    throw new Meteor.Error('list.unauthorized',
      'This list doesn\'t belong to you.');
  }

  return Lists.find(listId, {
    fields: {
      name: 1,
      incompleteCount: 1,
      userId: 1
    }
  });
});

// #2: Good! When the owner of the list changes, the old owner won't see it anymore
Meteor.publish('list', function (listId) {
  check(listId, String);

  return Lists.find({
    _id: listId,
    userId: this.userId
  }, {
    fields: {
      name: 1,
      incompleteCount: 1,
      userId: 1
    }
  });
});
```

In the first example, if the `userId` property on the selected list changes, the query in the publication will still return the data, since the security check in the beginning will not re-run. In the second example, we have fixed this by putting the security check in the returned query itself.

Unfortunately, not all publications are as simple to secure as the example above. For more tips on how to use `reywood:publish-composite` to handle reactive changes in publications, see the data loading article.

Passing options

For certain applications, for example pagination, you'll want to pass options into the publication to control things like how many documents should be sent to the client. There are some extra considerations to keep in mind for this particular case.

1. **Passing a limit**: In the case where you are passing the `limit` option of the query from the client, make sure to set a maximum limit. Otherwise, a malicious client could request too many documents at once, which could raise performance issues.
2. **Passing in a filter**: If you want to pass fields to filter on because you don't want all of the data, for example in the case of a search query, make sure to use MongoDB `$and` to intersect the filter coming from the client with the documents that client should be allowed to see. Also, you should whitelist the keys that the client can use to filter - if the client can filter on secret data, it can run a search to find out what that data is.
3. **Passing in fields**: If you want the client to be able to decide which fields of the collection should be fetched, make sure to intersect that with the fields that client is allowed to see, so that you don't accidentally send secret data to the client.

In summary, you should make sure that any options passed from the client to a publication can only restrict the data being requested, rather than extending it.

Served files

Publications are not the only place the client gets data from the server. The set of source code files and static assets that are served by your application server could also potentially contain sensitive data:

1. Business logic an attacker could analyze to find weak points.
2. Secret algorithms that a competitor could steal.
3. Secret API keys.

Secret server code

While the client-side code of your application is necessarily accessible by the browser, every application will have some secret code on the server that you don't want to share with the world.

Secret business logic in your app should be located in code that is only loaded on the server. This means it is in a `server/` directory of your app, in a package that is only included on the server, or in a file inside a package that was loaded only on the server.

If you have a Meteor Method in your app that has secret business logic, you might want to split the Method into two functions - the optimistic UI part that will run on the client, and the secret part that runs on the server. Most of the time, putting the entire Method on the server doesn't result in the best user experience. Let's look at an example, where you have a secret algorithm for calculating someone's MMR (ranking) in a game:

```
// In a server-only file, for example /imports/server/mmr.js
export const MMR = {
  updateWithSecretAlgorithm(userId) {
    // your secret code here
  }
}

// In a file loaded on client and server
Meteor.users.methods.updateMMR = new ValidatedMethod({
  name: 'Meteor.users.methods.updateMMR',
  validate: null,
  run() {
    if (this.isSimulation) {
      // Simulation code for the client (optional)
    } else {
      const { MMR } = require('/imports/server/mmr.js');
      MMR.updateWithSecretAlgorithm(this.userId);
    }
  }
});
```

Note that while the Method is defined on the client, the actual secret logic is only accessible from the server. Keep in mind that code inside `if (Meteor.isServer)` blocks is still sent to the client, it is just not executed. So don't put any secret code in there.

Secret API keys should never be stored in your source code at all, the next section will talk about how to handle them.

Securing API keys

Every app will have some secret API keys or passwords:

1. Your database password.
2. API keys for external APIs.

These should never be stored as part of your app's source code in version control, because developers might copy code around to unexpected places and forget that it contains secret keys. You can keep your keys separately in Dropbox, LastPass, or another service, and then reference them when you need to deploy the app.

You can pass settings to your app through a *settings file* or an *environment variable*. Most of your app settings should be in JSON files that you pass in when starting your app. You can start your app with a settings file by passing the `--settings` flag:

```
# Pass development settings when running your app locally
meteor --settings development.json

# Pass production settings when deploying your app to Galaxy
meteor deploy myapp.com --settings production.json
```

Here's what a settings file with some API keys might look like:

```
{
  "facebook": {
    "appId": "12345",
    "secret": "1234567"
  }
}
```

In your app's JavaScript code, these settings can be accessed from the variable `Meteor.settings`.

Read more about managing keys and settings in the Deployment article.

Settings on the client

In most normal situations, API keys from your settings file will only be used by the server, and by default the data passed in through `--settings` is only available on the server. However, if you put data under a special key called `public`, it will be available on the client. You might want to do this if, for example, you need to make an API call from the client and are OK with users knowing that key. Public settings will be available on the client under `Meteor.settings.public`.

API keys for OAuth

For the `accounts-facebook` package to pick up these keys, you need to add them to the service configuration collection in the database. Here's how you do that:

First, add the `service-configuration` package:

```
meteor add service-configuration
```

Then, upsert into the `ServiceConfiguration` collection:

```
ServiceConfiguration.configurations.upsert({
  service: "facebook"
}, {
  $set: {
    appId: Meteor.settings.facebook.appId,
    loginStyle: "popup",
    secret: Meteor.settings.facebook.secret
  }
});
```

Now, `accounts-facebook` will be able to find that API key and Facebook login will work properly.

SSL

This is a very short section, but it deserves its own place in the table of contents.

Every production Meteor app that handles user data should run with SSL.

Yes, Meteor does hash your password or login token on the client before sending it over the wire, but that only prevents an attacker from figuring out your password - it doesn't prevent them from logging in as you, since they could send the hashed password to the server to log in! No matter how you slice it, logging in requires the client to send sensitive data to the server, and the only way to secure that transfer is by using SSL. Note that the same issue is present when using cookies for authentication in a normal HTTP web application, so any app that needs to reliably identify users should be running on SSL.

Setting up SSL

- On Galaxy, configuration of SSL is automatic. See the help article about SSL on Galaxy.
- If you are running on your own infrastructure, there are a few options for setting up SSL, mostly through configuring a proxy web server. See the articles: Josh Owens on SSL and Meteor, SSL on Meteorpedia, and Digital Ocean tutorial with an Nginx config.

Forcing SSL Generally speaking, all production HTTP requests should go over HTTPS, and all WebSocket data should be sent over WSS.

It's best to handle the redirection from HTTP to HTTPS on the platform which handles the SSL certificates and termination.

- On Galaxy, enable the "Force HTTPS" setting on a specific domain in the "Domains & Encryption" section of the application's "Settings" tab.
- Other deployments *may* have control panel options or may need to be manually configured on the the proxy server (e.g. HAProxy, nginx, etc.). The articles linked above provide some assistance on this.

In the event that a platform does not offer the ability to configure this, the `force-ssl` package can be added to the project and Meteor will attempt to intelligently redirect based on the presence of the `x-forwarded-for` header.

HTTP Headers

HTTP headers can be used to improve the security of apps, although these are not a silver bullet, they will assist users in mitigating more common attacks.

Recommended: Helmet

Although there are many great open source solutions for setting HTTP headers, Meteor recommends Helmet. Helmet is a collection of 12 smaller middleware functions that set HTTP headers.

First, install helmet.

```
meteor npm install helmet --save
```

By default, Helmet can be used to set various HTTP headers (see link above). These are a good starting point for mitigating common attacks. To use the default headers, users should use the following code anywhere in their server side meteor startup code.

> Note: Meteor has not extensively tested each header for compatibility with Meteor. Only headers listed below have been tested.

```
// With other import statements
import helmet from "helmet";

// Within server side Meter.startup()
WebApp.connectHandlers.use(helmet())
```

At a minimum, Meteor recommends users to set the following headers. Note that code examples shown below are specific to Helmet.

Content Security Policy

> Note: Content Security Policy is not configured using Helmet's default header configuration.

From MDN, Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware.

It is recommended that users use CSP to protect their apps from access by third parties. CSP assists to control how resources are loaded into your application.

By default, Meteor recommends unsafe inline scripts and styles are allowed, since many apps typically use them for analytics, etc. Unsafe eval is disallowed, and the only allowable content source is same origin or data, except for connect which allows anything (since meteor apps make websocket connections to a lot of different origins). Browsers will also be told not to sniff content types away from declared content types.

```
// With other import statements
import helmet from "helmet";

// Within server side Meter.startup()
WebApp.connectHandlers.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "'unsafe-inline'"],
      connectSrc: ["*"],
      imgSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'"],
    }
  })
);
```

Helmet supports a large number of directives, users should further customise their CSP based on their needs. For more detail please read the following guide: Content Security Policy. CSP can be complex, so in addition there are some excellent tools out there to help, including Google's CSP Evaluator, Report-URI's CSP Builder, CSP documentation from Mozilla and CSPValidator.

The following example presents a potential CSP and other Security Headers used in a Production Meteor Application. This configuration may require customization, depending on your setup and use-cases.

```
/* global __meteor_runtime_config__ */
import { Meteor } from 'meteor/meteor'
import { WebApp } from 'meteor/webapp'
import { Autoupdate } from 'meteor/autoupdate'
import { check } from 'meteor/check'
import crypto from 'crypto'
import helmet from 'helmet'

const self = '\'self\''
const data = 'data:'
const unsafeEval = '\'unsafe-eval\''
const unsafeInline = '\'unsafe-inline\''
const allowedOrigins = Meteor.settings.allowedOrigins

// create the default connect source for our current domain in
// a multi-protocol compatible way (http/ws or https/wss)
const url = Meteor.absoluteUrl()
const domain = url.replace(/http(s)*:\/\//, '').replace(/\/$/, '')
const s = url.match(/(?!=http)s(?=:\/\/)/) ? 's' : ''
const usesHttps = s.length > 0
const connectSrc = [
  self,
  `http${s}://${domain}`,
  `ws${s}://${domain}`
]

// Prepare runtime config for generating the sha256 hash
// It is important, that the hash meets exactly the hash of the
// script in the client bundle.
// Otherwise the app would not be able to start, since the runtimeConfigScript
// is rejected __meteor_runtime_config__ is not available, causing
// a cascade of follow-up errors.
const runtimeConfig = Object.assign(__meteor_runtime_config__, Autoupdate, {
  // the following lines may depend on, whether you called Accounts.config
  // and whether your Meteor app is a "newer" version
  accountsConfigCalled: true,
  isModern: true
})

// add client versions to __meteor_runtime_config__
Object.keys(WebApp.clientPrograms).forEach(arch => {
  __meteor_runtime_config__.versions[arch] = {
    version: Autoupdate.autoupdateVersion || WebApp.clientPrograms[arch].version(),
    versionRefreshable: Autoupdate.autoupdateVersion || WebApp.clientPrograms[arch].versionR
    versionNonRefreshable: Autoupdate.autoupdateVersion || WebApp.clientPrograms[arch].versi
    // comment the following line if you use Meteor < 2.0
    versionReplaceable: Autoupdate.autoupdateVersion || WebApp.clientPrograms[arch].versionR
  }
})

const runtimeConfigScript = `__meteor_runtime_config__ = JSON.parse(decodeURIComponent("${en
const runtimeConfigHash = crypto.createHash('sha256').update(runtimeConfigScript).digest('ba

const helpmentOptions = {
  contentSecurityPolicy: {
    blockAllMixedContent: true,
    directives: {
      defaultSrc: [self],
      scriptSrc: [
        self,
        // Remove / comment out unsafeEval if you do not use dynamic imports
        // to tighten security. However, if you use dynamic imports this line
        // must be kept in order to make them work.
        unsafeEval,
        `'sha256-${runtimeConfigHash}'`
      ],
      childSrc: [self],
      // If you have external apps, that should be allowed as sources for
      // connections or images, your should add them here
      // Call helmetOptions() without args if you have no external sources
      // Note, that this is just an example and you may configure this to your needs
      connectSrc: connectSrc.concat(allowedOrigins),
      fontSrc: [self, data],
      formAction: [self],
      frameAncestors: [self],
      frameSrc: ['*'],
      // This is an example to show, that we can define to show images only
      // from our self, browser data/blob and a defined set of hosts.
      // Configure to your needs.
      imgSrc: [self, data, 'blob:'].concat(allowedOrigins),
      manifestSrc: [self],
      mediaSrc: [self],
      objectSrc: [self],
      // these are just examples, configure to your needs, see
      // https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/
      sandbox: [
        // allow-downloads-without-user-activation // experimental
        'allow-forms',
        'allow-modals',
        // 'allow-orientation-lock',
        // 'allow-pointer-lock',
        // 'allow-popups',
        // 'allow-popups-to-escape-sandbox',
        // 'allow-presentation',
        'allow-same-origin',
        'allow-scripts',
        // 'allow-storage-access-by-user-activation ', // experimental
        // 'allow-top-navigation',
        // 'allow-top-navigation-by-user-activation'
      ],
      styleSrc: [self, unsafeInline],
      workerSrc: [self, 'blob:']
    }
  },
  // see the helmet documentation to get a better understanding of
  // the following configurations and settings
  strictTransportSecurity: {
    maxAge: 15552000,
    includeSubDomains: true,
    preload: false
  },
  referrerPolicy: {
    policy: 'no-referrer'
  },
  expectCt: {
    enforce: true,
    maxAge: 604800
  },
  frameguard: {
    action: 'sameorigin'
  },
  dnsPrefetchControl: {
    allow: false
  },
  permittedCrossDomainPolicies: {
    permittedPolicies: 'none'
  }
}

// We assume, that we are working on a localhost when there is no https
// connection available.
// Run your project with --production flag to simulate script-src hashing
if (!usesHttps && Meteor.isDevelopment) {
  delete helpmentOptions.contentSecurityPolicy.blockAllMixedContent;
  helpmentOptions.contentSecurityPolicy.directives.scriptSrc = [
    self,
    unsafeEval,
    unsafeInline,
  ];
}

// finally pass the options to helmet to make them apply
helmet(helpmentOptions)
```

X-Frame-Options

> Note: The X-Frame Options header is configured using Helmet's default header configuration.

From MDN, the X-Frame-Options HTTP response header can be used to indicate whether or not a browser should be allowed to render a page in a `<frame>`, `<iframe>` or `<object>`. Sites can use this to avoid clickjacking attacks, by ensuring that their content is not embedded into other sites.

Meteor recommend users configure the X-Frame-Options header for same origin only. This tells browsers to prevent your webpage from being put in an iframe. By using this config, you will set your policy where only web pages on the same origin as your app can frame your app.

With Helmet, Frameguard sets the X-Frame-Options header.

```
// With other import statements
```

```
import helmet from "helmet";

// Within server side Meter.startup()
WebApp.connectHandlers.use(helmet.frameguard());  // defaults to sameorigin
```

For more detail please read the following guide: Frameguard.

Security checklist

This is a collection of points to check about your app that might catch common errors. However, it's not an exhaustive list yet—if we missed something, please let us know or file a pull request!

1. Make sure your app doesn't have the `insecure` or `autopublish` packages.
2. Validate all Method and publication arguments, and include the `audit-argument-checks` to check this automatically.
3. Apply rate limiting to your application to prevent DDoS attacks.
4. Deny writes to the `profile` field on user documents.
5. Use Methods instead of client-side insert/update/remove and allow/deny.
6. Use specific selectors and filter fields in publications.
7. Don't use raw HTML inclusion in Blaze unless you really know what you are doing.
8. Make sure secret API keys and passwords aren't in your source code.
9. Secure the data, not the UI - redirecting away from a client-side route does nothing for security, it's a nice UX feature.
10. Don't ever trust user IDs passed from the client. Use `this.userId` inside Methods and publications.
11. Set up secure HTTP headers using Helmet, but know that not all browsers support it so it provides an extra layer of security to users with modern browsers.
12. At the end of the day, Meteor is a Node.js app so make sure to also follow the best practises to ensure maximum security.

App Protection

App Protection on Galaxy Hosting is a feature in our proxy server layer that sits in front of every request to your application. This means that all requests across servers are analyzed and measured against expected limits. This will help protect against DoS and DDoS attacks that aimed to overload servers and make your app unavailable for legitimate requests.

If a type of request is classified as abusive (we're not going to go into the specifics as to how we determine this), we will stop sending these requests to your app, and we start to return HTTP 429 (Too Many Requests).*

Although not all attacks are preventable, our App Protection functionality, along with standard AWS protection in front of our servers, will provide a greater level of security for all applications deployed to Galaxy moving forward.

For additional security, it is best to configure your app to limit the messages received via WebSockets, as our proxy servers are only acting in the first connection and not in the WebSocket messages after the connection is established. Meteor has the DDP Rate Limiter configuration already available, find out more here.