Playing with Proxy

(And Reflect)

(But Reflect Does Not Alliterate)

(So It Kinda Ruins the Title)

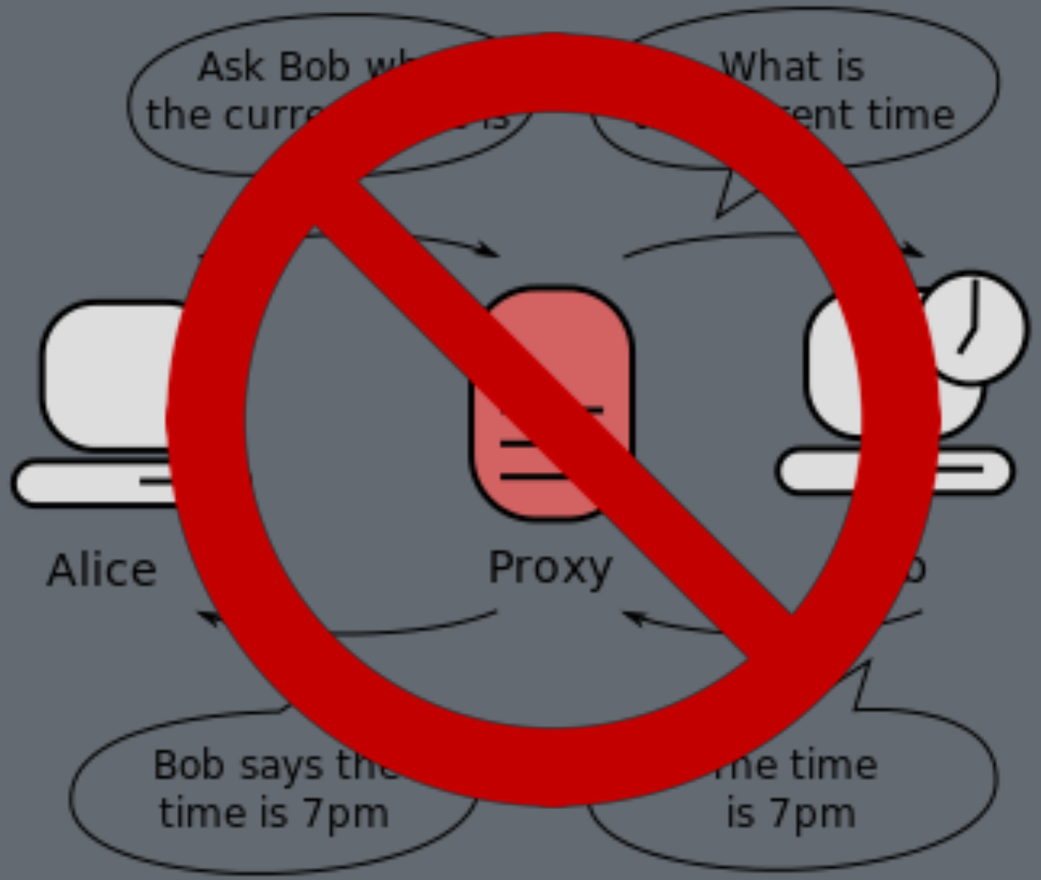
Hello, I'm Ehden

node.js agent engineer @ Contrast Security

@ehden (CCJS slack)
github.com/cixel
ehdens@gmail.com
ehden@contrastsecurity.com

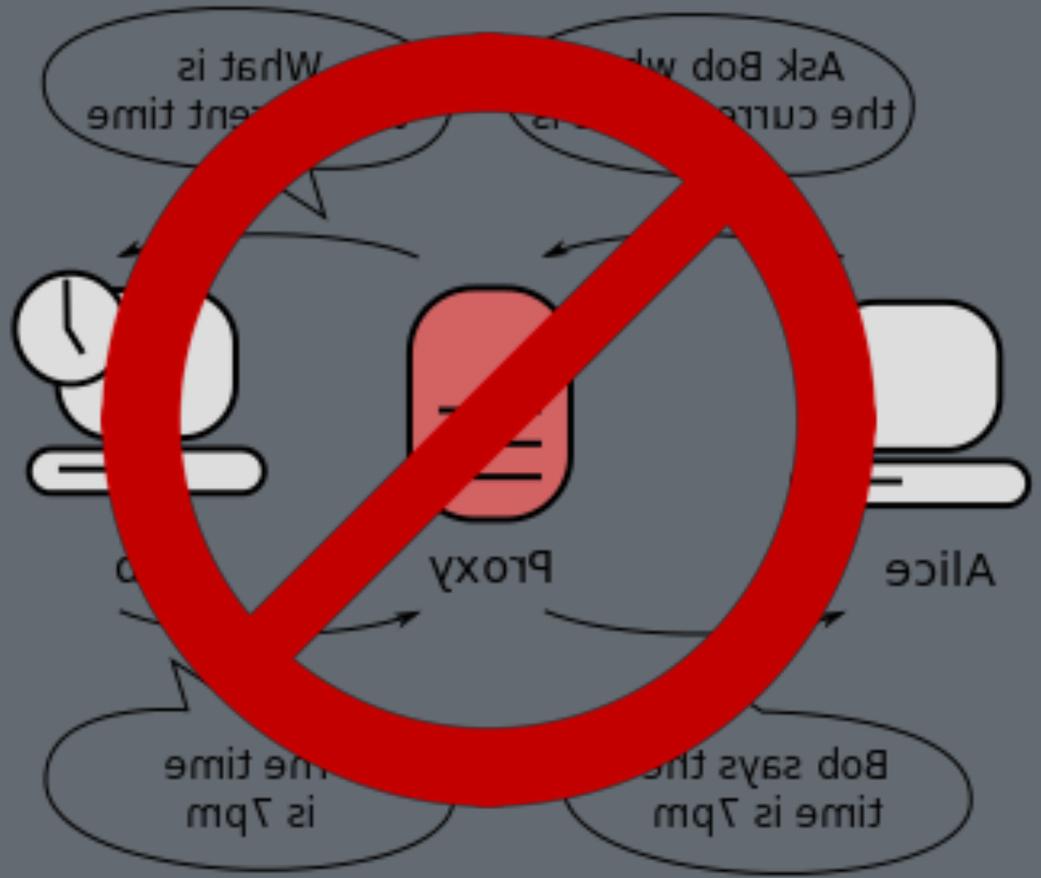
Proxy

Used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc).[1]



[1] MDN

Reflect



Reflect is a built-in object that provides methods for interceptable JavaScript operations. The methods are the same as those of proxy handlers. Reflect is not a function object, so it's not constructible.[2]

[2] MDN

Terminology 📖

target: the object being wrapped (*must* be an object)

handler: object which holds traps

```
const proxy = new Proxy(
    {
        foo: 'bar'
    },
    {
        get(tar, prop, recv) { /* ... */ },
        set(tar, prop, val) { /* ... */ },
        // ...
    }
);
```

Terminology 📖

target: the object being wrapped (*must* be an object)

handler: object which holds traps

```
const proxy = new Proxy(
    {
        foo: 'bar'
    },
    {
        get(tar, prop, recv) { /* ... */ },
        set(tar, prop, val) { /* ... */ },
        // ...
    }
);
```

Terminology 📖

target: the object being wrapped (*must* be an object)

handler: object which holds traps

```
const proxy = new Proxy(
    {
        foo: 'bar'
    },
    {
        get(tar, prop, recv) { /* ... */ },
        set(tar, prop, val) { /* ... */ },
        // ...
    }
);
```

Terminology 📖

trap: method providing intercept behavior for an operation

- `apply()`
- `construct()`
- `defineProperty()`
- `deleteProperty()`
- `get()`
- `getOwnPropertyDescriptor()`
- `getPrototypeOf()`
- `has()`
- `isExtensible()`
- `ownKeys()`
- `preventExtensions()`
- `set()`
- `setPrototypeOf()`

Terminology 📖

invariant: condition which must be satisfied by a trap

```
const s = new String('hello');

const p = new Proxy(s, {
    get(target, property, receiver) {
        return 'fish';
    }
});


console.log(p[0]);
// TypeError: 'get' on proxy: property '0' is read-only and you tried to do something PRETTY silly there, friend
```

```
const person = new Person('Jean-Luc Picard');

const handler = {
    get(target, property, receiver) {
        if (property === 'name') {
            return 'Locutus of Borg';
        }

        return Reflect.get(...arguments);
    },

    set(target, property, value) {
        if (property === 'name') {
            // don't actually set
            console.log('Resistance is futile.');
            return true
        }
        return Reflect.set(...arguments);
    }
};

const borg = new Proxy(person, handler);
```

```
const person = new Person('Jean-Luc Picard');

const handler = {
    get(target, property, receiver) {
        if (property === 'name') {
            return 'Locutus of Borg';
        }

        return Reflect.get(...arguments);
    },

    set(target, property, value) {
        if (property === 'name') {
            // don't actually set
            console.log('Resistance is futile.');
            return true
        }
        return Reflect.set(...arguments);
    }
};

const borg = new Proxy(person, handler);
```

```
const person = new Person('Jean-Luc Picard');

const handler = {
    get(target, property, receiver) {
        if (property === 'name') {
            return 'Locutus of Borg';
        }

        return Reflect.get(...arguments);
    },

    set(target, property, value) {
        if (property === 'name') {
            // don't actually set
            console.log('Resistance is futile.');
            return true
        }
        return Reflect.set(...arguments);
    }
};

const borg = new Proxy(person, handler);
```

```
const person = new Person('Jean-Luc Picard');

const handler = {
    get(target, property, receiver) {
        if (property === 'name') {
            return 'Locutus of Borg';
        }

        return Reflect.get(...arguments);
    },

    set(target, property, value) {
        if (property === 'name') {
            // don't actually set
            console.log('Resistance is futile.');
            return true
        }
        return Reflect.set(...arguments);
    }
};

const borg = new Proxy(person, handler);
```

```
const person = new Person('Jean-Luc Picard');

const handler = {
    get(target, property, receiver) {
        if (property === 'name') {
            return 'Locutus of Borg';
        }

        return Reflect.get(...arguments);
    },

    set(target, property, value) {
        if (property === 'name') {
            // don't actually set
            console.log('Resistance is futile.');
            return true
        }
        return Reflect.set(...arguments);
    }
};

const borg = new Proxy(person, handler);
```

```
const person = new Person('Jean-Luc Picard');

const handler = {
    get(target, property, receiver) {
        if (property === 'name') {
            return 'Locutus of Borg';
        }

        return Reflect.get(...arguments);
    },

    set(target, property, value) {
        if (property === 'name') {
            // don't actually set
            console.log('Resistance is futile.');
            return true
        }
        return Reflect.set(...arguments);
    }
};

const borg = new Proxy(person, handler);
```

"Star Trek TNG is cool so Proxy should be added to the spec"

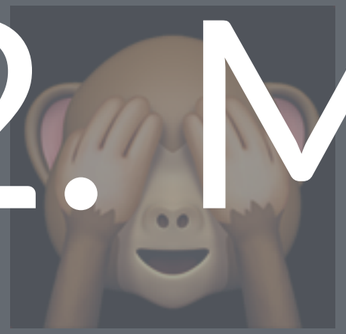
--W3C, Ecma, Brendan Eich, and Patrick Stewart

(all at the same time and in weird, hive-mind-like unison)

1. Logging, Spying, Validating

```
const p = new Proxy({}, {
    get(tar, prop, recv) {
        console.log(`getting ${prop}`);
        return Reflect.get(...arguments);
    }

    set(tar, prop, value) {
        console.log(`setting ${prop}`);
        return Reflect.set(...arguments);
    }
});

p.a = '!'; // setting a
```

2. Monkey Patching



the "old" way

```
class MyClass {
    constructor() {}

    someFunction() {
        // do something
    }
}

const someFn = MyClass.prototype.someFunction;
MyClass.prototype.someFunction = function() {
    doSomething(this, arguments);
    const result = someFn.apply(this, arguments);
    doSomethingElse(this, arguments, result);
};
```

using Proxy/Reflect

```
class MyClass {
    constructor() {}

    someFunction() {
        // do something
    }
}

MyClass.prototype.someFunction = new Proxy(MyClass.prototype.someFunction, {
    apply(target, thisArg, args) {
        doSomething(thisArg, args);
        const result = Reflect.apply(...arguments);
        doSomethingElse(thisArg, args, result);
    }
})
```

3. Dynamic APIs

```
const obj = {};

function APIify(obj) {
    const handler = {
        get(tar, prop, recv) {
            if (prop.startsWith('get')) {
                const name = prop.substring(3);
                name[0] = name[0].toLowerCase();
                return function() {
                    // make some REST request?
                }
            }
        }
    };
    return new Proxy(obj, handler);
}

const A = APIify(obj);
A.getSomeStuff();
```

4. Revoking Access to an Object

Proxy.revocable()

```
const obj = { a: 'test' };

const revocable = Proxy.revocable(obj, {
    // ...
});

revocable.revoke();

console.log(obj.a); // TypeError
```

Partially Revocable Proxies

```
class Fickle {
    constructor() { this.revoked = false; }

    proxy(obj) {
        const handler = {
            get: (tar, prop, recv) => {
                console.log(this);
                if (this.revoked) throw new TypeError('nah');

                return Reflect.get(tar, prop, recv);
            }
        };
        return new Proxy(obj, handler);
    }

    revoke() { this.revoked = true; }
    restore() { this.revoked = false; }
}

const fickle = new Fickle();
const revokable = fickle.proxy({ a: 'test' });
fickle.revoke();
console.log(revokable.a); // TypeError: nah
```

5. Bodyless Functions

```
const f = new Proxy(function() {}, {
    apply(t, thisArg, args) {
        function actualFunction() {
            console.log('hi');
        }

        return Reflect.apply(actualFunction, thisArg, args);
    }
});

f(); // hi
console.log(f.toString()); // 'function () {}'
```

6. Recursive Proxies

```
const handler = {
    get(tar, prop, recv) {
        const result = Reflect.get(...arguments);

        if (result && typeof result === 'object') {
            return new Proxy(result, handler);
        }

        return result;
    }
}
```

```
const handler = {
    get(tar, prop, recv) {
        const result = Reflect.get(...arguments);

        if (result && typeof result === 'object') {
            return new Proxy(result, handler);
        }

        return result;
    }
}
```

```
const handler = {
    get(tar, prop, recv) {
        const result = Reflect.get(...arguments);

        if (result && typeof result === 'object') {
            return new Proxy(result, handler);
        }

        return result;
    }
}
```

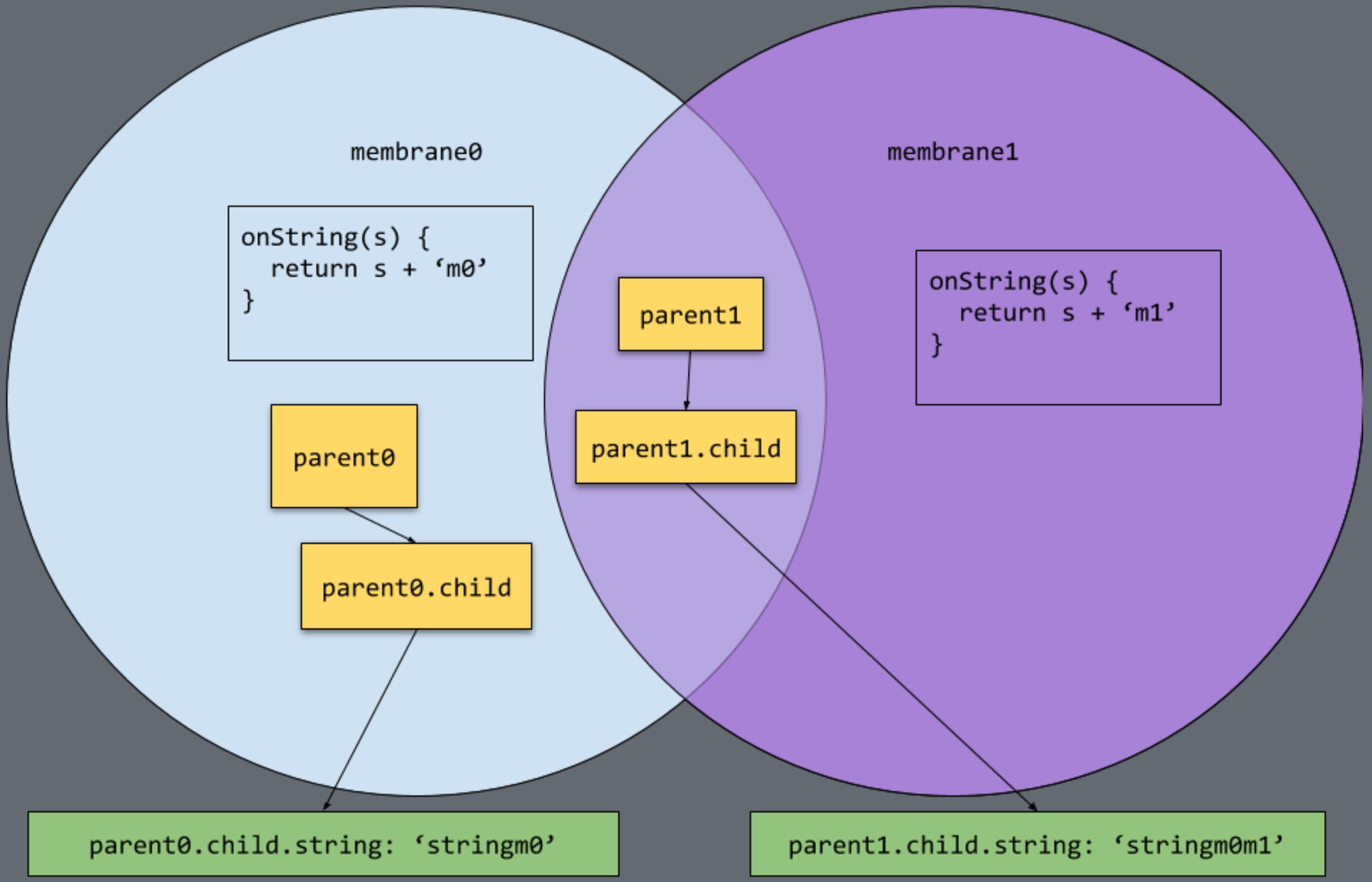
```
const handler = {
    get(tar, prop, recv) {
        const result = Reflect.get(...arguments);

        if (result && typeof result === 'object') {
            return new Proxy(result, handler);
        }

        return result;
    }
}
```

7. Proxy Membranes

```
class Membrane {
    constructor() {
        // map wrapped --> Mapping
        this.mappings = new WeakSet();
    }

    wrap(obj) {
        const handler = makeHandler(this);

        return new Proxy(obj, handler);
    }

    includes(val) { return this.mappings.has(val); }

    getMapping(t) { return this.mappings.get(t); }

    // "base case"; do whatever you want in here!
    onPrimitive(p) { return p; }
}
```

```
/**
 * Mapping formalizes the association between wrapped and unwrapped versions of objects in the Membrane.
 */
class Mapping {
  constructor(orig, wrapped) {
    /** original (fully unwrapped). equals target if this is the only membrane the orig belongs to. */
    this.orig = orig;
    /** proxy */
    this.wrapped = wrapped;
  }
}
```

```
function makeHandler(membrane) {
    const handler = {
        get(tar, prop, recv) => {
            const result = Reflect.get(...arguments);

            if (membrane.includes(tar)) {
                return membrane.getMapping(tar).wrapped;
            }

            if (result && typeof result === 'object') {
                const wrapped = membrane.wrap(result);
                const mapping = new Mapping(target, wrapped);

                membrane.mappings.set(target, mapping);
                membrane.mappings.set(wrapped, mapping);

                return wrapped;
            }

            return membrane.onPrimitive(p);
        }
    };

    return handler;
}
```


membrane0
onString(s) {
return s + ‘m0’
}
parent0
parent0.child
membrane1
parent1
parent1.child
onString(s) {
return s + ‘m1’
}
parent0.child.string: ‘stringm0’
parent1.child.string: ‘stringm0m1’

Others!

Others!

- specifying default values for fields on objects you don't control creation of

Others!

- specifying default values for fields on objects you don't control creation of
- cascading property changes - flip a switch to change several values

Others!

- specifying default values for fields on objects you don't control creation of
- cascading property changes - flip a switch to change several values
- make arrays accessible as objects

This is all super dope, so why isn't Proxy used everywhere?

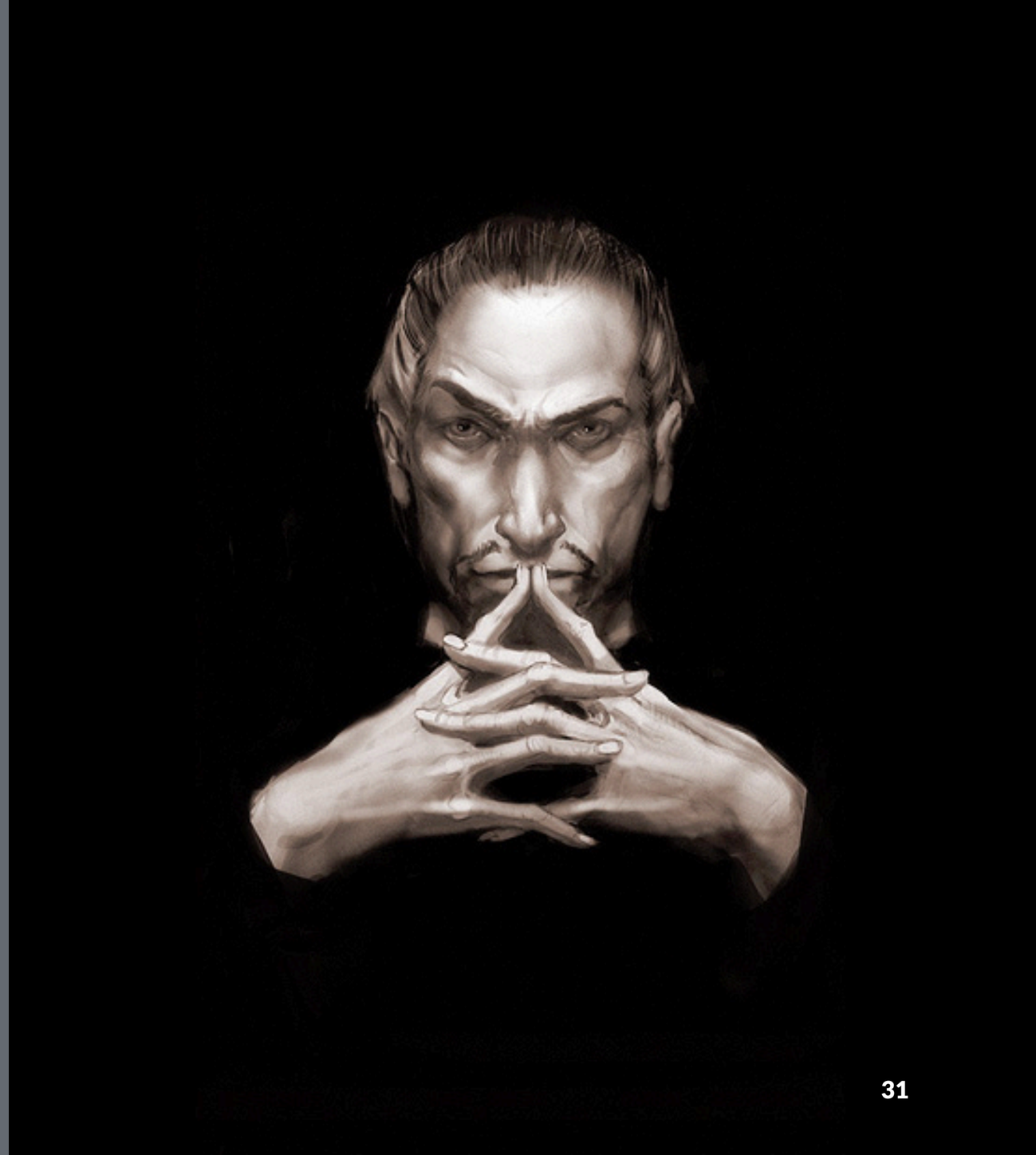
Transparency

“And these are your reasons, my lord?"

"Do you think I have others?" said Lord Vetinari. "My motives, as ever, are entirely transparent."

Hughnon reflected that 'entirely transparent' meant either that you could see right through them or that you couldn't see them at all.

— Terry Pratchett, The Truth



The End!

Contact:

- @ehden (CCJS slack)
- github.com/cixel
- ehdens@gmail.com
- ehden@contrastsecurity.com

Presentation Materials:

- github.com/cixel/charmcityjs-apr2019

Image Credits:

- Contrast Security
- H2g2bob [CC0]
- Borg
- Lord Vetinari by juliedillon
- Join the Team