Web Application Security

BellevueJS - October 2018

whoami

- Andrew Hoffman, Product Security Engineer @ Salesforce
    - Lead for Lightning Platform UI Teams

- Prior: Software Security Engineer @ Salesforce
    - Engineer on Locker Service Security Library
    - Worked with TC39 on “Realms” proposal

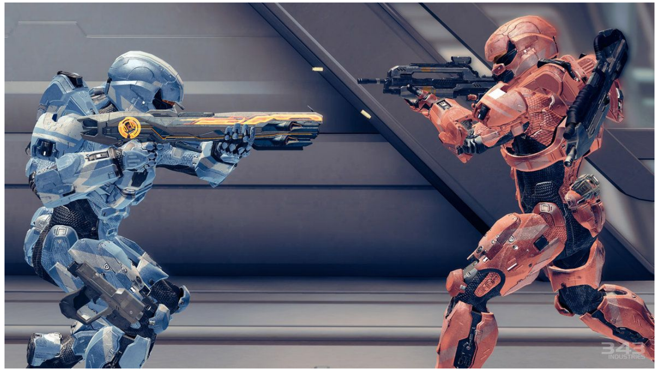
- Ancient History: JavaScript Developer
    - EmberJS / AngularJS / Aura and NodeJS

Target Audience

- This is an entry level security primer
- Aimed at existing software developers
- Background required: JS, DOM, HTTP

Schedule

- Begin on the offensive
  - How can we map a web app?
  - What are common attack vectors?
  - How can we turn those into exploits?
- Conclude on the defensive
  - How do we defend our app against threats?
  - How can we structure our organization to mitigate risk?



Offense Overview

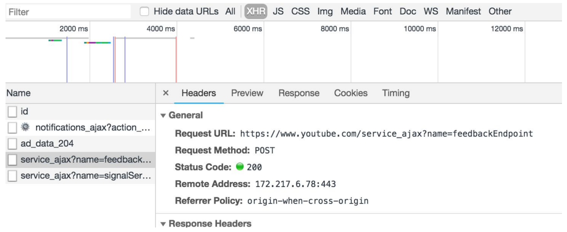
The web was not designed as an application platform.

At its inception, the web was actually just a platform for sharing static documents.

As a result, early hackers exploited primarily via network and server level attacks.

Todays web applications are much more vulnerable than their predecessors as a result of their ability to not only read data on a server, but to write and modify data on the server as well.

Almost all of today’s major breaches are as a result of a web application being configured to read and operate on input from a user or third party developer.



id
notifications_ajax?action_...
ad_data_204
service_ajax?name=feedback...
service_ajax?name=signalSer...

▾ General

Request URL: `https://www.youtube.com/service_ajax?name=feedbackEndpoint`
Request Method: `POST`
Status Code: ● `200`
Remote Address: `172.217.6.78:443`
Referrer Policy: `origin-when-cross-origin`

▾ Response Headers

Mapping a Web App

The first stage in hacking a web application is building a map of its data schemas, user interface and API's.

We can map out the features available to us by walking through it and monitoring network requests.

After that we want to do some behind-the-scenes research to get a better understanding of the web app's APIs.

1. Find all subdomains
2. Find all endpoints
3. Find what HTTP methods the endpoints support
4. Fuzz the endpoints (inject random data into the endpoints and see what is returned)

Finding Subdomains

- Check Public Records: `dig http://bbburger.com -t any`
- Attempt a “Zone Transfer Attack”: `dig http://bbburger.com -t axfr`
    - This is a very simple attack where we basically ask other nameservers to give us DNS records by pretending we are a DNS server ourselves. It only works if the server is not configured to reject suspicious looking DNS servers.
- Attempt a “Reverse DNS Lookup”: `dig -x <ip-address-of-bbburger>`
    - Yes, some servers accidentally specify their own subdomains in the DNS records.
- Brute Force Subdomains
    - This can be done via a script, or a more advanced tool like DNSRecon
        - Scripts can do this by using a dictionary of common subdomains, making a request and seeing if a response is provided (error responses work fine!)

Discovering Endpoints

Once you have discovered all of the domains and subdomains related to a URL you want to figure out the API endpoints.

- You can brute force using a dictionary of common endpoint names
  - Once you figure out the structure of the API it becomes much easier
    - REST helps with this
    - AKA: GET /users/<user-id> means there might be GET /burgers/<burger-id>
- You can also check search engines and directories for links to the app:
  - site:http://www.bbburger.com <some-string>
    - The results of the search will give you a number of GET requests into the app
- Social networks and news articles may also contain valuable links to examine

Brute Forcing HTTP Methods

Once you have a list of subdomains and endpoints, you can take your existing endpoint/subdomain combinations and attempt to make HTTP requests using different HTTP verbs and payloads:

GET /users/<user-id> -> POST /users/<user-id>

After covering all of your surface area from the subdomain and endpoint discovery phase, you now have a map of the web application without ever having visited it.

Fuzzing API Endpoints

After generating a map of a web application you wish to hack, the next steps are fuzzing the endpoints to determine what type of data the endpoint supports and how it is sanitized.

You want to start by doing this in an automated fashion, until you understand the payloads the API expects.

After you know that an API expects a hash in the form { user, password, realm } for example you can begin crafting more targeted logic attacks manually.

You now can hack the web app without even having access to the UI or a user account!

Attack Vectors

An attack vector is a method by which an attacker targets a web app seeking a vulnerability to exploit.

These come in many shapes and target many parts of an app’s technology stack.

Because this presentation is targeted at JavaScript developers, all example source code used throughout will be JavaScript.

Do note: most of these exploits can be replicated in other languages, client technology and server frameworks.

Common Attack Vectors

- XSS
- Poor Authentication Implementation
- Misconfigured Access Control
- CSRF
- OSS Libraries and Package Managers

There are of course many more attack vectors to choose from, in fact too many to cover in one presentation.

OWASP - a non profit software security research institution is an excellent starting point for exploring new methodologies for breaking into web applications.

Cross Site Scripting (XSS)

XSS is an attack where the attacker crafts their own script (usually in JavaScript), and deploys it to an app they do not have ownership of in a method that causes it to be executed when other users visit the app or perform a specific action.

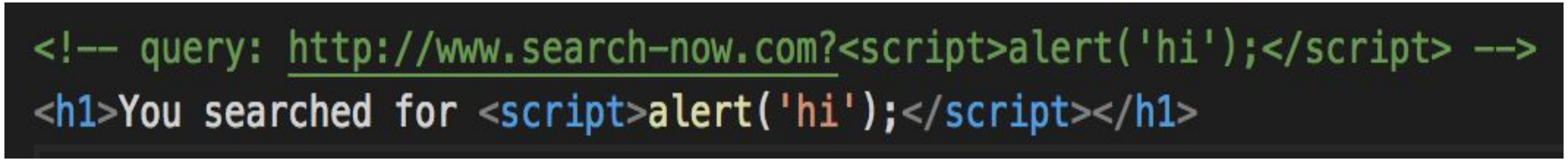
XSS attacks are often categorized into three major types:

- Reflected
- Stored
- DOM Based

Reflected XSS

Reflected XSS is a type of attack where script is not stored in a database, but instead is reflected upon the user of an app.

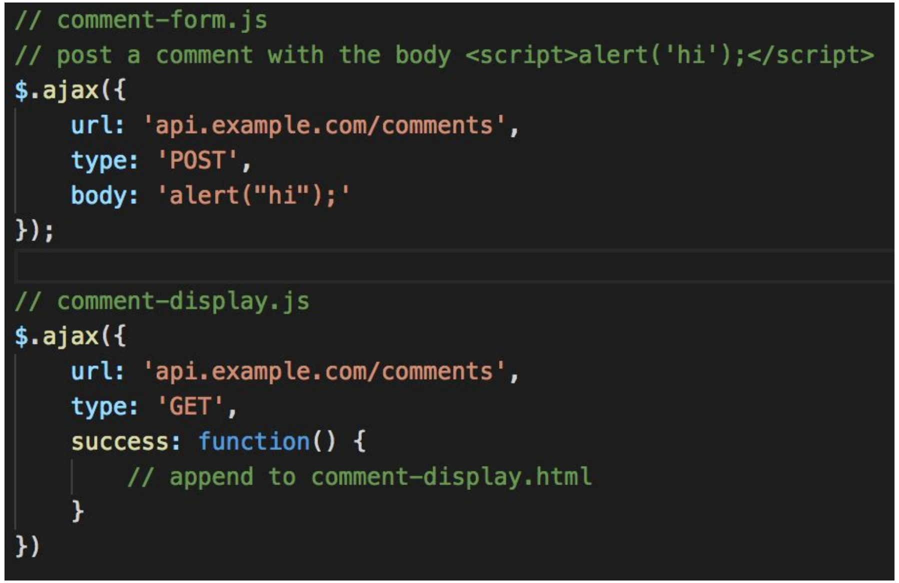
A common example of this is a search engine that takes a string as a query, and than navigates to a results page. In the results page, if the user’s query string is injected directly into the DOM without sanitization it is possible that the user’s query string could be evaluated as script.



```
<!-- query: http://www.search-now.com?<script>alert('hi');</script> -->
<h1>You searched for <script>alert('hi');</script></h1>
```

Stored XSS

Similar in function to a reflected XSS, stored XSS differs in that the script itself is stored in a database. This often makes it more dangerous than a reflected XSS as it will generally affect a larger number of users.



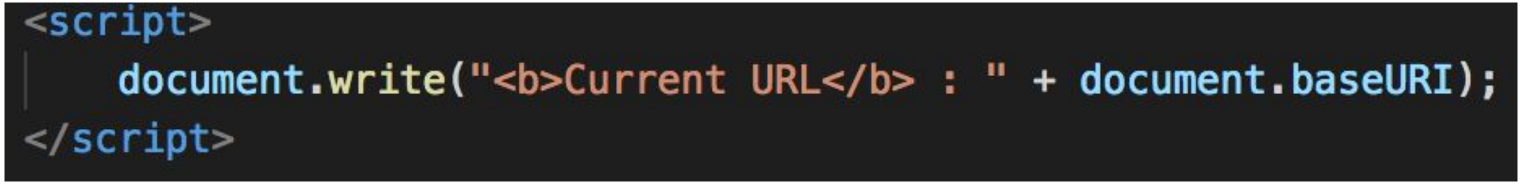
```
// comment-form.js
// post a comment with the body <script>alert('hi');</script>
$.ajax({
    url: 'api.example.com/comments',
    type: 'POST',
    body: 'alert("hi");'
});

// comment-display.js
$.ajax({
    url: 'api.example.com/comments',
    type: 'GET',
    success: function() {
        // append to comment-display.html
    }
})
```

DOM XSS

DOM XSS attacks are an advanced XSS that rely on deep knowledge of the browser DOM and the API’s supported. Often these attacks may target only one major browser and can often be due to an improper implementation of the DOM spec.

Often these employ utilizing a browser DOM method (aka a “sink”) to bypass filtration on the client. Common targets are: **eval**, **setTimeout**, **setInterval**, **document.write** and **element.innerHTML**. Even more advanced users have found success using API’s like **DOMParser** and **document.implementation**.



```
<script>
    document.write("<b>Current URL</b> : " + document.baseURI);
</script>
```

Poor Authentication Implementation

- What error messages are displayed on sign-up / sign-in?
  - “Username is already taken”
- Are authentication routes rate limited?
- If resetting a password with a username, is the email exposed?
- Do session tokens expire?
- Is any unique user identifier sequentially generated?
- Are authentication pages crawled by search engines?

Misconfigured Access Control

Let’s assume your app has two tiers of user account: “admin” and “default”.

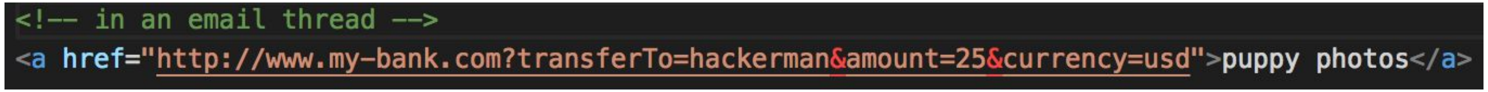
- **How does your server decide who is an admin, versus a default user?**
  - If the server relies on the client sending an auth token back, can the server tell if the auth token was modified?
- **For admin-specific pages on the client, can default users access them by spoofing session state in cookies?**
  - Gaining access to admin UI is great for mapping functionality even if none of the commands work.
- **When a user sends a request to update their profile, what fields are permitted for update?**
  - Are the permitted fields in a whitelist or a blacklist?

Cross-Site Request Forgery (CSRF)

CSRF attacks are attacks where a hacker is able to get a user to perform an action on their behalf.

Typically the hacker can’t actually see the result of these actions, so instead the hacker will target ways of changing the application state.

For example, transfering funds or promoting the hacker to an admin user.

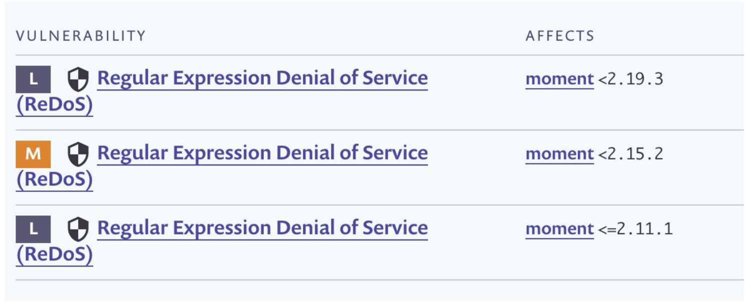


```html
<!-- in an email thread -->
<a href="http://www.my-bank.com?transferTo=hackerman&amount=25&currency=usd">puppy photos</a>
```

OSS Libraries and Package Managers

As a result of the web application ecosystem being so dependent on OSS, in particular with NPM and Yarn - it's important to frequently scan your codebase for known vulnerabilities.

Many OSS vulnerability databases exist on the web, for example: https://snyk.io



VULNERABILITY	AFFECTS
L Regular Expression Denial of Service (ReDoS)	moment <2.19.3
M Regular Expression Denial of Service (ReDoS)	moment <2.15.2
L Regular Expression Denial of Service (ReDoS)	moment <=2.11.1

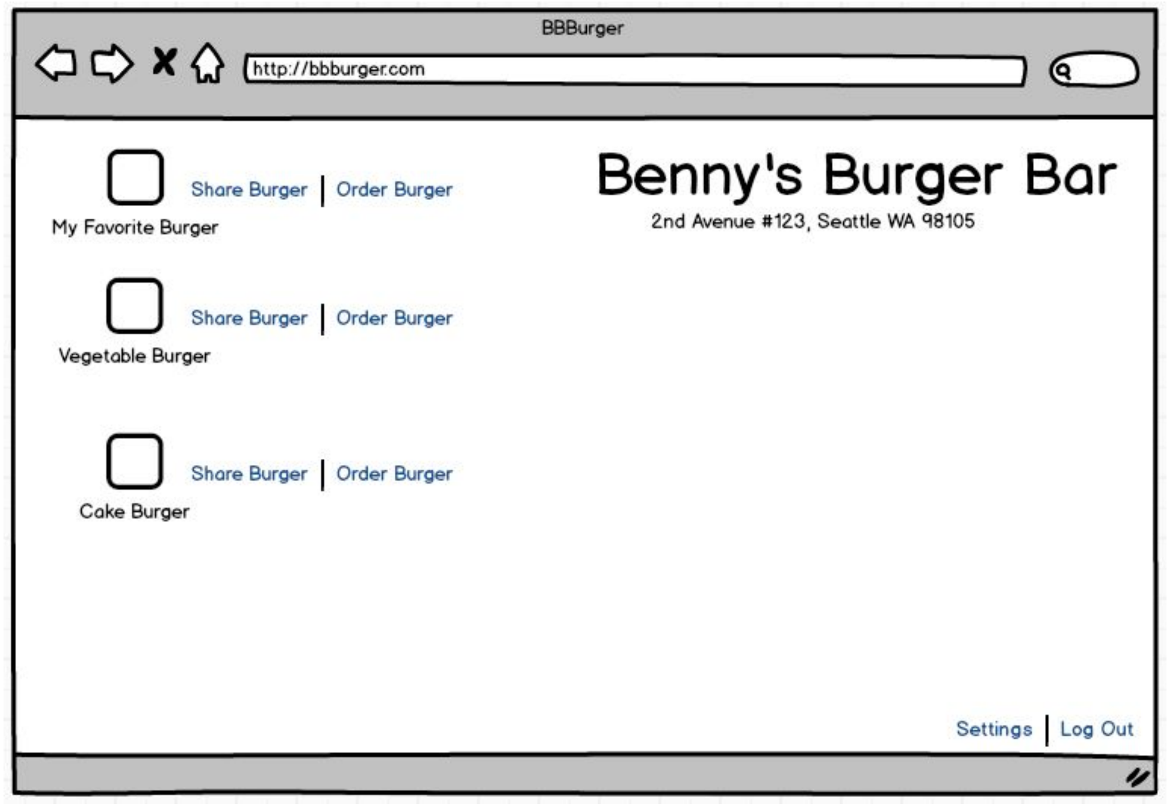
Example: Benny's Burger Bar

Benny's Burger Bar is a small, local restaurant and bar offering custom made burgers.

They have a web application located at http://bbburger.com

Web app features:

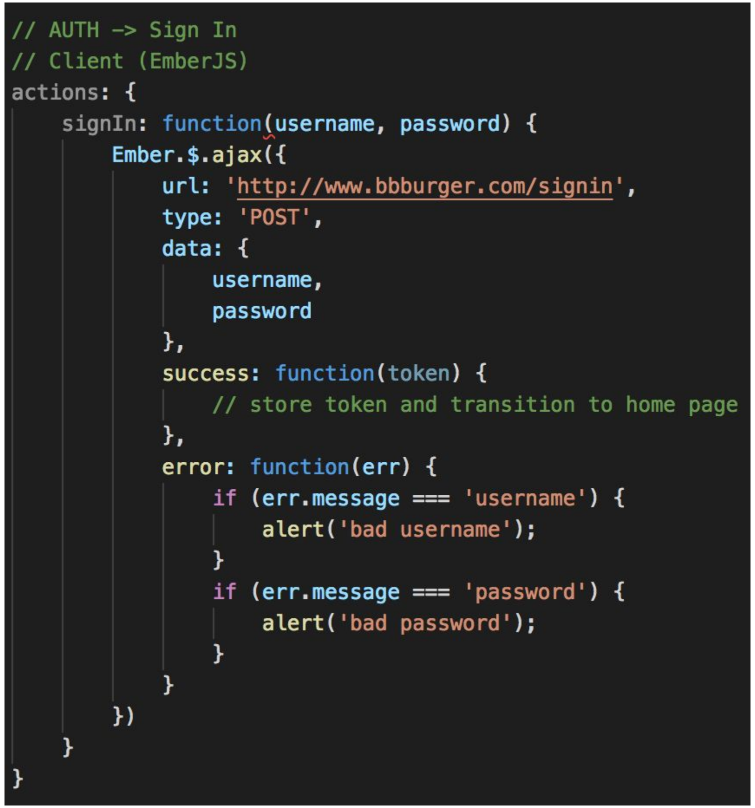
- Authentication
- Storing burger recipes
- Sharing burgers recipes with your friends


BBBurger
http://bbburger.com
Share Burger | Order Burger
My Favorite Burger
Share Burger | Order Burger
Vegetable Burger
Share Burger | Order Burger
Cake Burger
Benny's Burger Bar
2nd Avenue #123, Seattle WA 98105
Settings | Log Out

Sign In

- First off, the sign in request is sending usernames and passwords over HTTP - meaning they are in plain text while in transit.
    - Wifi Pineapples
    - 99% of apps should force HTTPS traffic, because the performance hit is usually minimal.
- Next, the server is sending an error message back to the client which reflects if the username was incorrect or the password was incorrect.
    - This is a common mistake web applications make, because it allows hackers to easily probe for valid user accounts.

This demonstrates a poor implementation of an authentication system.



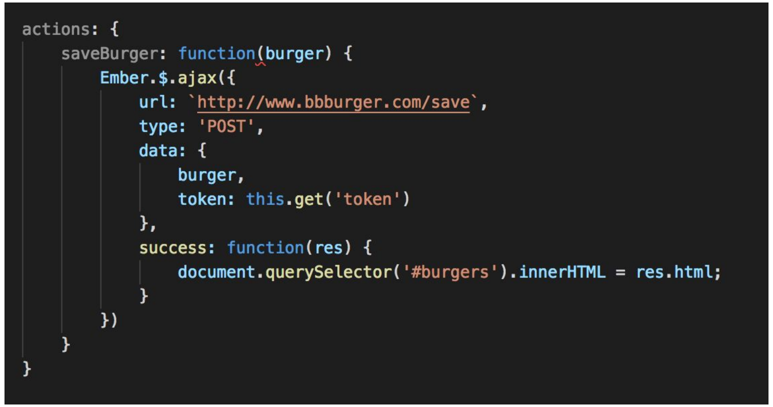
```
// AUTH -> Sign In
// Client (EmberJS)
actions: {
    signIn: function(username, password) {
        Ember.$.ajax({
            url: 'http://www.bbburger.com/signin',
            type: 'POST',
            data: {
                username,
                password
            },
            success: function(token) {
                // store token and transition to home page
            },
            error: function(err) {
                if (err.message === 'username') {
                    alert('bad username');
                }
                if (err.message === 'password') {
                    alert('bad password');
                }
            }
        })
    }
}
```

Save Burger

This is an example of a stored XSS which is much more common than you would expect.

Let's assume the user used a form with the structure: { title, ingredients }

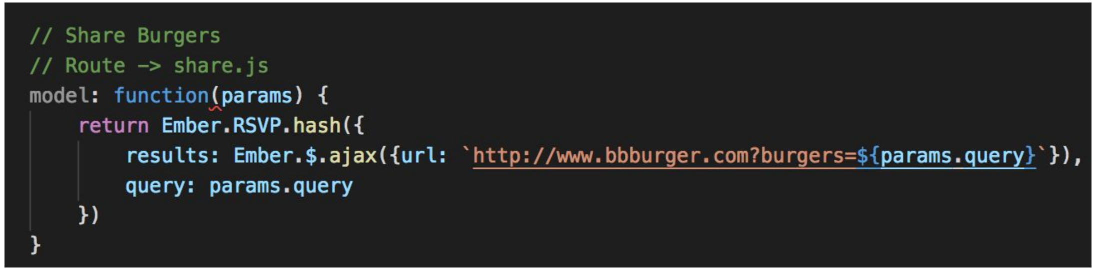
If the title of this burger was <script>alert('hi');</script>, and the server did not provide proper sanitization before wrapping it in HTML - than the alert() call would be executed immediately upon it's injection into the DOM via innerHTML.



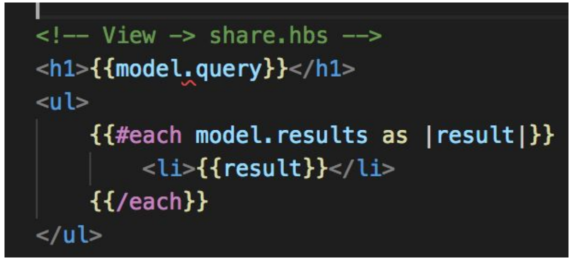
```
actions: {
    saveBurger: function(burger) {
        Ember.$.ajax({
            url: `http://www.bbburger.com/save`,
            type: 'POST',
            data: {
                burger,
                token: this.get('token')
            },
            success: function(res) {
                document.querySelector('#burgers').innerHTML = res.html;
            }
        })
    }
}
```

Sharing Burgers

This is an example of a reflected XSS, where a user could provide a link on another service that includes script inside of it's query param.



```
// Share Burgers
// Route -> share.js
model: function(params) {
    return Ember.RSVP.hash({
        results: Ember.$.ajax({url: `http://www.bbburger.com?burgers=${params.query}`}),
        query: params.query
    })
}
```



```
<!-- View -> share.hbs -->
<h1>{{model.query}}</h1>
<ul>
    {{#each model.results as |result|}}
        <li>{{result}}</li>
    {{/each}}
</ul>
```

Advanced XSS Vectors

Executable script comes in many forms in the browser, not only inside of script tags. It is possible to use many built in browser DOM APIs to hide and execute script from inside of:

- SVG
- Blob
- XML
- PDF
- XMLDocument, HTMLDocument

And many, many more.

Defense Overview

Understanding how to hack a web app is essential to understanding how to secure a web app.

While most developers consider use cases of legitimate users when architecting features, we need to also consider the use cases of illegitimate users.

We must also consider that users may attempt to bypass our client, so validation at the client level is often not enough. It may be useful for legitimate users, but not capable of protecting the server.

Defending against XSS

- Input should be sanitized at three different levels
  - Server
  - Client
  - Database
- When sanitizing input, use a whitelist model instead of a blacklist
- When appending to the DOM, user element.innerText instead of element.innerHTML
- Avoid use of DOM API’s that convert text into script or DOM nodes, unless you have a good use case for them: DOMParser, document.implementation, eval
  - Some apps have legitimate use cases for these, as they are very powerful tools.
    - They should only be used when there is no alternative.

Secure Authentication

- Set expiration on tokens
- Enforce HTTPS
- Configure cookies
  - httpOnly attribute -> prevent JavaScript from being able to read cookies
  - secure attribute -> don’t send cookies unless over HTTPS
- Don’t show errors that give away data
  - “Bad username” should instead be “Bad username or password” at minimum
- Rate limit sign up and sign in endpoints

Secure Access Control Mechanisms

- Check integrity of tokens to ensure they are not modified
- Whitelist fields allowed to be updated by the user
- Run access control modification APIs only on a local server
    - For example, don’t expose PHPMyAdmin over the network

CSRF Defense

- Prevent writeable state from being set via URL
- Add additional authentication steps (e.g. 2FA) for mission critical actions
- Verify “origin” header in requests
  - Indicates where FETCH originates from
- Verify “referer” header in requests
  - Indicates where a hyperlink originated from
- Consider CAPTCHA for mission critical events

The Defensive Mindset

Writing secure, defensive code will mitigate risk but not eliminate it.

The larger your codebase grows, the more potential risk is present. Risk grows exponentially with complexity.

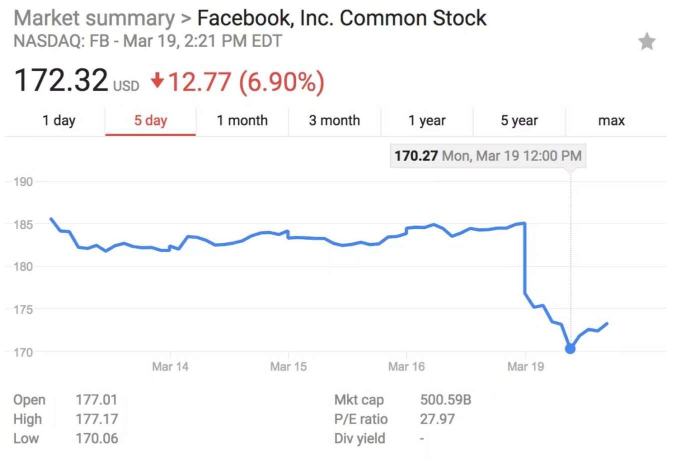
- **Consider a combination of approaches to best mitigate risk:**
    - Write secure, defensive oriented code.
        - Integrate security into your culture, and let developers know they have a large stake in the security of your product.
    - Hire third party penetration testers to attempt to break into your app.
    - Build a bug bounty program, so hackers are incentivized to sell vulnerabilities to you rather than the black market.

Writing Secure Code

- Integrate SSDL into your development lifecycle
    - Begin new products with architecture and data flow diagrams.
    - Begin new features with threat models.
        - Both threat models and architecture/dfd should be reviewed with a security engineer and obvious threats should be mitigated prior to development.
- Teach developers about common threats, and allow them to code defensively against them.
    - Convince management that their goals are not only feature development and performance, but security as well.
        - This must be a cultural switch, as many development organizations see security as a bottleneck.

Third Party Penetration Testers

It is often much cheaper for a large software company to hire penetration testers than to fend of a large data breach.


Market summary > Facebook, Inc. Common Stock
NASDAQ: FB - Mar 19, 2:21 PM EDT

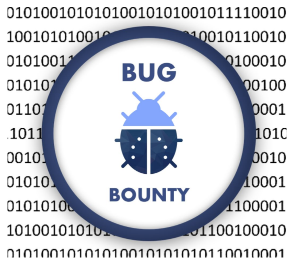
172.32 USD ↓12.77 (6.90%)

Open	177.01	Mkt cap	500.59B
High	177.17	P/E ratio	27.97
Low	170.06	Div yield	-

Facebook lost nearly 40b USD in market capitalization as a result of a major breach this year.

- If Facebook had hired a single team of quality penetration testers to validate the data exposed by the "View As" API, the data exposed would have most likely been found and at a cost of **20-40k USD**.
- Breaches like this also leave long lasting brand damage, and in particular hurt B2B companies who might lose major customers.

Bug Bounty Programs



IBM released a study that claims the average cost of a data breach in 2018 is about **3.5 million USD**.

- By offering a legal alternative to selling vulnerabilities on the black market, you may mitigate serious breaches.
    - **The going rate for a P0 vulnerability disclosure is between 15k and 30k. That is significantly cheaper than 3.5 million dollars in damage from a P0 vulnerability being exploited.**
- Companies like BugCrowd and HackerOne can help companies set up legal bug bounty programs with contracts that protect white hat hackers and incentivize them to help you find vulnerabilities.
- These programs can also be useful for testing your hacking abilities.

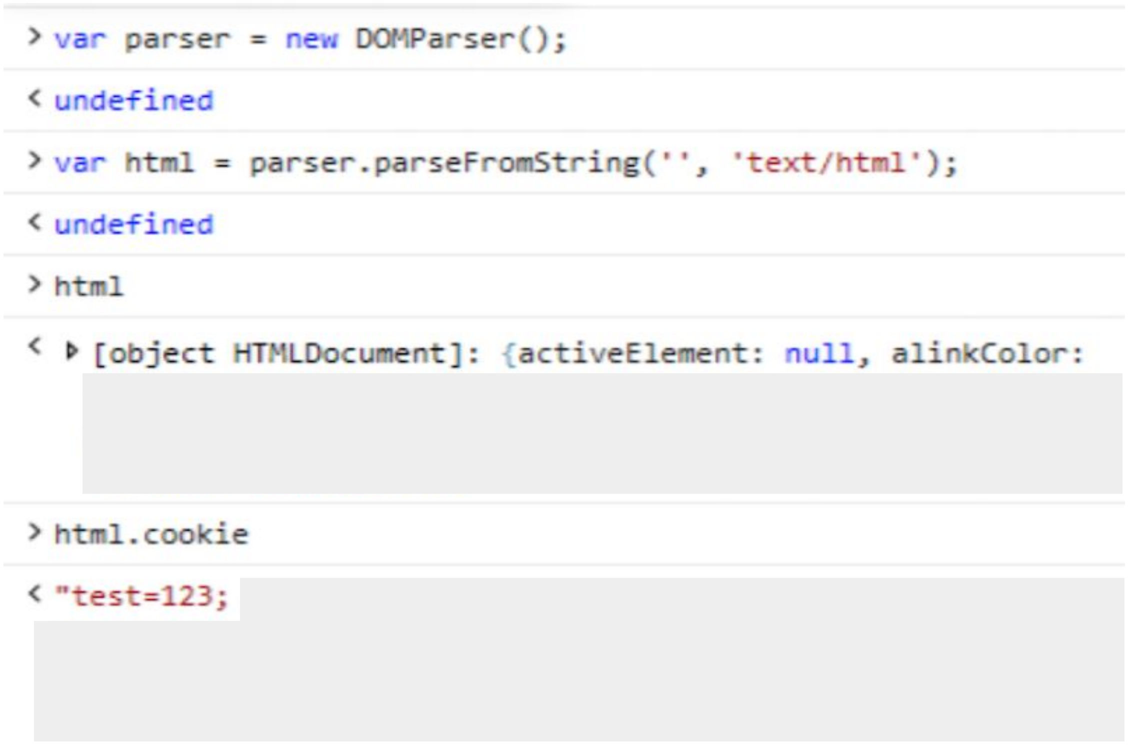
Example Bug Bounty: Microsoft Edge Bug Cookies

Here is an example of a bug bounty submission I made to the Edge team.

It explains how Edge’s DOM model for “document” was improperly implemented and does not follow the spec correctly.

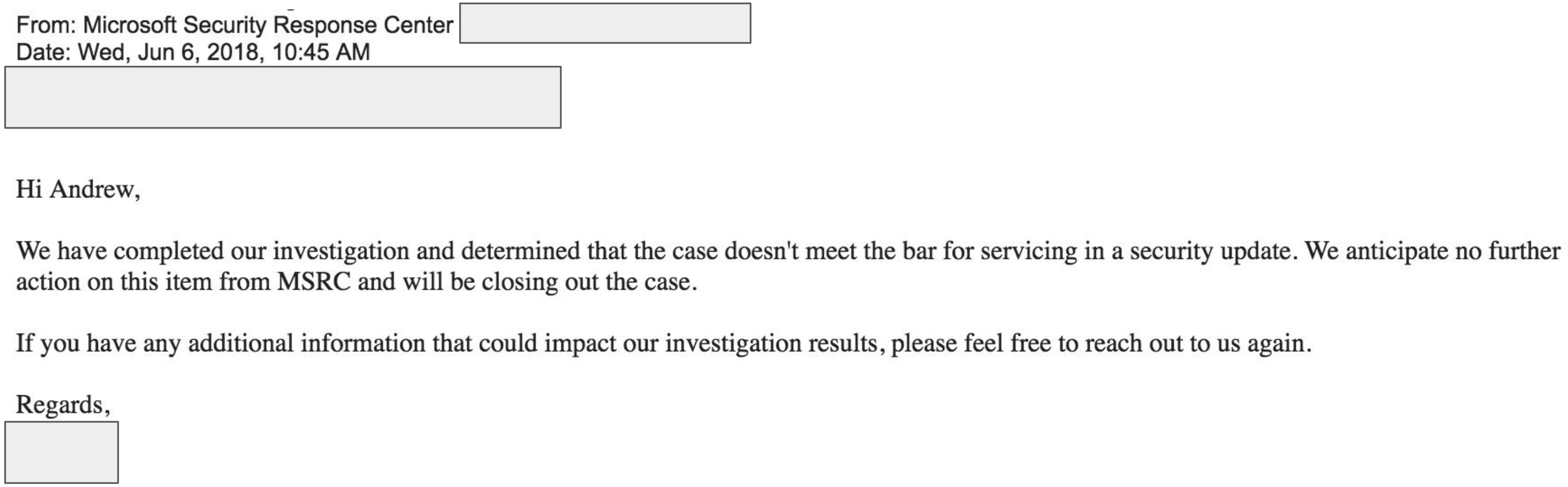
I was able to view session tokens in multiple apps that allow third party JS as a result of this.

https://www.andrewhoffman.me/cookie-bypass-edge/



```
> var parser = new DOMParser();
< undefined
> var html = parser.parseFromString('', 'text/html');
< undefined
> html
< ▸ [object HTMLDocument]: {activeElement: null, alinkColor:
```
[redacted]
```
> html.cookie
< "test=123;
```
[redacted]

Results of Bug Bounty Submission



From: Microsoft Security Response Center [redacted]
Date: Wed, Jun 6, 2018, 10:45 AM
[redacted]

Hi Andrew,

We have completed our investigation and determined that the case doesn't meet the bar for servicing in a security update. We anticipate no further action on this item from MSRC and will be closing out the case.

If you have any additional information that could impact our investigation results, please feel free to reach out to us again.

Regards,
[redacted]

Conclusion

- Building a secure software company requires a combination of both offensive and defensive knowledge.
- Developers are an integral part of building a secure software product. Security cannot be an afterthought to development.
- Security culture must run deep in your company, **a company is only as secure as its weakest links**.
- There are ways of mitigating risk, for example offering bug bounties to ethical hackers and hiring third party firms to evaluate your features from an outsider’s perspective.