

# Introduction to System Security

IPADS, Shanghai Jiao Tong University

<https://www.sjtu.edu.cn>



# Attacks Happen

## Lots of personal info stolen

- Attackers broke into server w/ ~800K records on Utah patients

## Phishing attacks

- Users at ORNL tricked by phishing email about benefits from HR

## Millions of PCs are under control of an adversary

- Which is called "Botnet"

## Stuxnet

- Infected Iran's uranium processing facilities
- Fake certificate



FORTUNE

SUBSCRIBE

JUNE 1, 2015

Ellen Pao appeals sexism case she lost to VC firm

NEWS

POPULAR

VIDEOS

FORTUNE 500



TECH AT&amp;T

AT&T workers stole almost 280,000 customers' personal data: FCC APRIL 8, 2015

Ellen Pao appeals sexism case she lost to VC firm 8:08 PM EDT

Google's workplace diversity still has a long way to go 7:33 PM EDT

KFC: Our chickens don't have eight legs 7:10 PM EDT

Why Facebook's algorithm matters: because 60% of millennials get news there 6:45 PM EDT

Cloud software company Coupa snags \$80 million to cut corporate spending 5:30 PM EDT

Le Mans, an Aston Martin Vantage GT12 and me: What's not to love? 5:26 PM EDT

Jerry made serious cash in the last season of 'Seinfeld' 4:48 PM EDT

Is EMC cloud strategy a victory for customer choice or just confusing?

# AT&T workers stole almost 280,000 customers' personal data: FCC

by Benjamin Snyder

@WriterSnyder

APRIL 8, 2015, 3:26 PM EDT



Tech

Finance

Politics

Strategy

Life

Sports

Video

All

Search

**MILITARY & DEFENSE**More: [Stuxnet](#) [Iran](#) [Israel](#) [Cyberwarfare](#)

# The Stuxnet Attack On Iran's Nuclear Plant Was 'Far More Dangerous' Than Previously Thought



MICHAEL B KELLEY



NOV. 20, 2013, 12:58 PM

63,680

11



FACEBOOK



LINKEDIN



TWITTER

**Popular Articles**

by Taboola



NFL quarterback who made \$12.5 million and retired at age 26 plans to relax and remodel his house



Here's Why This 'Shark Tank' Investor Says He Hated Mark Cuban For 2 Years

The Stuxnet virus that ravaged Iran's Natanz nuclear facility "was far more dangerous than the cyberweapon that is now



# In-flight Wi-Fi is 'direct link' to hackers

HACKING / 15 APRIL 15 / by MICHAEL RUNDLE



Aeroplanes with in-flight Wi-Fi are vulnerable to hacks by passengers and could be targeted by a "malicious attacker" on the ground, a US report has warned.

The US Government Accountability Office (GAO) described the potential dangers in a new report for the Federal Aviation Administration (FAA) titled *"FAA Needs a More Comprehensive Approach to Address Cybersecurity As Agency Transitions to NextGen"*.

The study explained that IP networks of all kinds left flights open to cyberattacks -- whether that's in-flight wireless entertainment systems, internet-based cockpit communications or the new Next Generation Air



A US report has warned that the presence of smartphones and tablets on aeroplanes had increased the risk of in-flight cyberattacks *Shutterstock*



Hey there!



Secure | <https://www.apple.com>

# Hey there!

This may or may not be the site you are looking for! This site is obviously affiliated with Apple, but rather a demonstration of a flaw in the way domains are handled in browsers.

[See what this is about](#)

Before I explain the details of the vulnerability, you should take a look at the [proof-of-concept](#).

[Punycode](#) makes it possible to register domains with foreign characters. It works by converting individual domain label to an alternative format using only ASCII characters. For example, the domain "xn-s7y.co" is equivalent to "短.co".

From a security perspective, Unicode domains can be problematic because many Unicode characters are difficult to distinguish from common ASCII characters. It is possible to register domains such as "xn-pple-43d.com", which is equivalent to "apple.com". It may not be obvious at first glance, but "apple.com" uses the Cyrillic "a" (U+0430) rather than the ASCII "a" (U+0061). This is known as a [homograph attack](#).

Fortunately modern browsers have mechanisms in place to limit IDN homograph attacks. The page [IDN in](#)



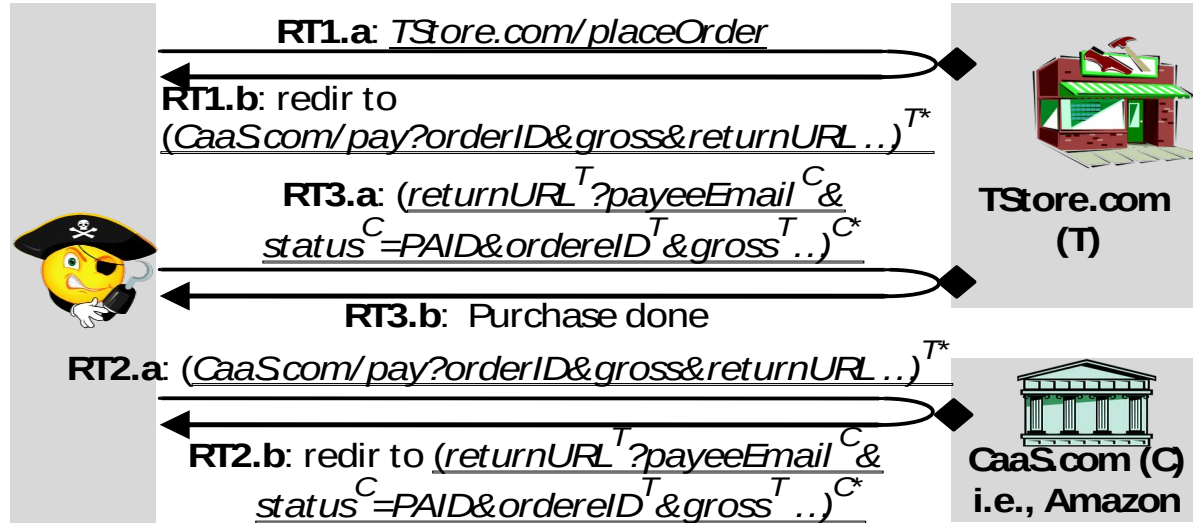
告知

公安网中病毒，下午停止办理业务。

2017年5月13日



# HOW TO SHOP FOR FREE ONLINE (S&P'11)



**TStore.com/placeOrder:** `orderId=InsertPendingOrder ()`

**TStore.com/finishOrder (handler of RT3.a):**

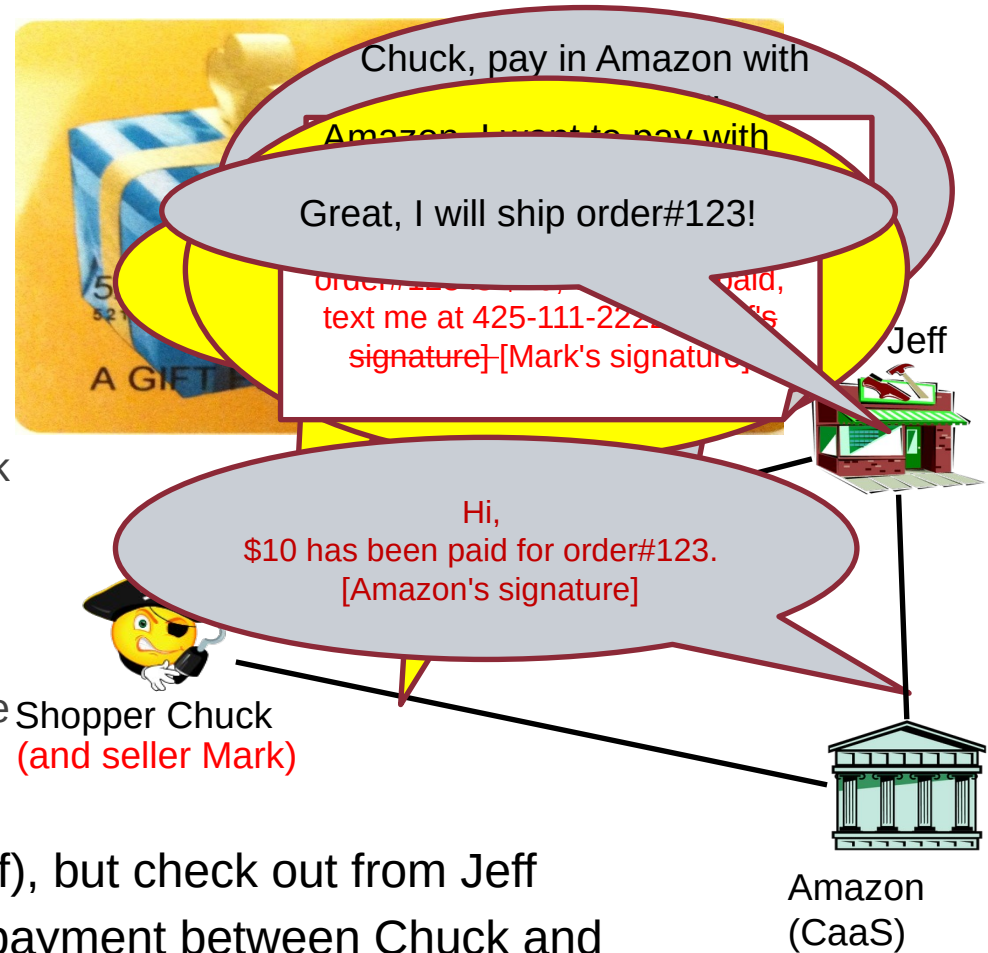
```

if (verifySignature(RT3.a) ≠ CaaS) exit;
if (GetMsgField("status") ≠ PAID) exit; /* payment status*/
order= GetOrderByID(orderID);
if (order=NULL or order.status ≠ PENDING) exit;
order.status=PAID;
    
```



# Flaw & exploit

- Anyone can register an Amazon seller account, so can Chuck.
  - We purchased a \$25 MasterCard gift card by cash
  - We registered it under the name "Mark Smith" with fake address/phone number
  - Registered for seller accounts in PayPal, Amazon and Google using the card
- Chuck's trick
  - Pay to Mark (i.e., Chuck himself), but check out from Jeff
  - Amazon is tricked to tell Jeff a payment between Chuck and Mark
  - Jeff is confused by Amazon





## Other Attacks

**Buffer overflow attack (stack/heap)**

**ROP attack (Return-Oriented Programming)**

**Password attack**

**Phishing attack**

**XSS attack (Cross Site Script)**

**SQL injection attack**

**Integer overflow attack**

**Social engineering attack**

**Side-channel attack**

**...**

# Security: Real World VS. Computer

## Security in general not a particularly new concern

- Banks, military, legal system, etc. have always worried about security

## Similarities with computer security:

- Want to compartmentalize (different keys for bike vs. safe deposit box)
- Log and audit for compromises
- Use legal system for deterrence

# Security: Real World VS. Computer

## Significant differences with computer security:

- Internet makes attacks fast, cheap, and scalable
  - Huge number of adversaries: bored teenagers, criminals worldwide, etc.
  - Anonymous adversaries: no strong identity on the internet
  - Adversaries have lots of resources (compromised PCs in a botnet)
  - Attacks can often be automated: systems compromised faster than can react
- Users often have poor intuition about computer security
  - E.g., misunderstand implications of important security decisions

# Why is Security So Hard?

## Security is a negative goal

- Want to achieve something despite whatever adversary might do

## A positive goal: "XYB can read exam.txt".

- Ask XYB to check if our system meets this positive goal

## A negative goal: "Ben cannot read exam.txt".

- Ask John if he can read exam.txt?
- Good to check, but not nearly enough..
- Must reason about all possible ways in which John might get the data

## How might Ben try to get the contents of exam.txt?

## Ways to Access exam.txt

Change permissions on exam.txt to get access

Access disk blocks directly

Access exam.txt via ipads.se.sjtu.edu.cn

Reuse memory after XYB's text editor exits, read data

Read backup copy of exam.txt from XYB's text editor

Intercept network packets to file server storing exam.txt

Send XYB a trojaned text editor that emails out the file

Steal disk from file server storing exam.txt

Get discarded printout of exam.txt from the trash

Call sysadmin, pretend to be XYB, reset his password

**... when should we stop thinking of more ways?**

# 学生期末将微信名改成教务处 私聊老师要考卷

2020年01月01日 17:31 西安晚报 作者：西安晚报

A<sup>-</sup> | A<sup>+</sup> |

原标题：学生期末将微信名改成教务处 私聊老师要考卷

又到期末考试季，你慌不慌？日前，@湖南中医药大学 的一位同学把自己微信名字改为湖南中医药大学教务处，并私聊自己的任课老师，让老师把考卷发给他，事后，被“要考卷”的老师在学校的群里直接公布了这一做法[允悲]，还不忘督促同学们“认真看书，不要瞎想”。

来源：北青网



# Why is Security So Hard?

## Security sometimes conflicts with other goals

- Security VS. performance
- Security VS. availability
- Security VS. fault tolerance
- Security VS. convenience
- Security VS. simplicity
- ...

# Fault Tolerant Fails Here

## Why not using fault tolerant techniques for security?

- Since security can be seen as a kind of fault

## Two reasons

- Result of such “fault” may be too much (user cannot afford once)
  - E.g., exam leakage, rocket launching
- Failures due to attack might be highly correlated
  - In fault-tolerance, usually assume independent failures

# Much Harder to Reason about Security

**No complete solution; we can't always secure every system**

- One solution: Trust nothing that you didn't create by yourself

**What are we going to learn?**

- How to model systems in the context of security
- How we think about and assess risks
- Techniques for assessing common risks
  - There are things we can do to make systems more secure
- Know trade-offs
  - Security vs. performance, security vs. convenience, and security vs. simplicity

## 2 Steps towards Building a More Secure System

How to make progress in building a secure system?

- Be clear about goals: "**policy**"
- Be clear about assumptions: "**threat model**"



## Policy: Goals

### Information security goals:

- **Confidentiality**: limit who can read data
- **Integrity**: limit who can write data

### Liveness goals:

- **Availability**: ensure service keeps operating

## Threat Model: Assumptions

**Often don't know in advance who might attack, or what they might do**

- Adversaries may have different goals, techniques, resources, expertise

**Cannot be secure against arbitrary adversaries**

- As we saw with Ben vs. XYB
- Adversary might be your hardware vendor, software vendor, administrator, ...
- Need to make some plausible assumptions to make progress

# Threat Model: Assumptions

## What does a threat model look like?

- Adversary controls some computers, networks (but not all)
- Adversary controls some software on computers he doesn't fully control
- Adversary knows some information, such as passwords or keys (but not all)
- Adversary knows about bugs in your software?
- Physical attacks?
- Social engineering attacks?
- Resources? (Can be hard to estimate either resources or requirements!)



# Threat Model: Assumptions

## Unrealistic / incomplete threat models

- Adversary is outside of the company network / firewall
- Adversary doesn't know legitimate users' passwords
- Adversary won't figure out how the system works

## Despite this, important to have a threat model

- Can reason about assumptions, evolve threat model over time
- When a problem occurs, can figure out what exactly went wrong and re-design

## Overly-ambitious threat models not always a good thing

- Not all threats are equally important
- Stronger requirements can lead to more complexity
- Complex systems can develop subtle security problems



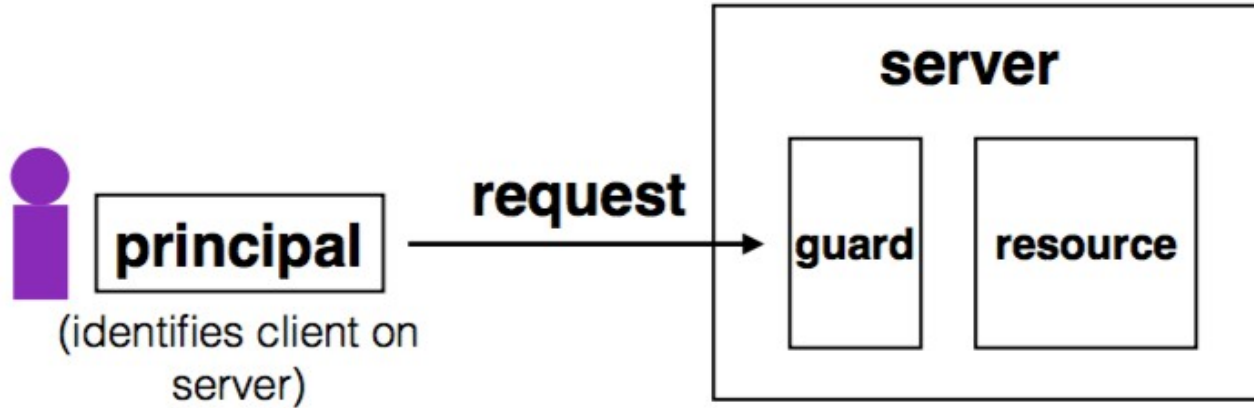
# GUARD MODEL

# Guard Model of Security

## Complete mediation

- Only way to access the resource involves the guard
- 1. Must enforce client-server modularity
  - Adversary should not be able to access server's resources directly
  - E.g., assume OS enforces modularity, or run server on separate machine
- 2. Must ensure server properly invokes the guard in all the right places

# Complete Mediation



Guard typically provides:

- **Authentication**: is the principal who they claim to be?
- **Authorization**: does principal have access to perform request on resource?

## Designing the Guard

### Two functions often provided by a guard:

- **Authentication:** request -> principal
  - E.g., client's username, verified using password
- **Authorization:** (request, principal, resource) -> allow?
  - E.g., consult access control list (ACL) for resource

### Simplifies security

- Can consider the guards under threat model
- But don't forget about complete mediation

## Example: Unix FS

**Resource:** files, directories.

**Server:** OS kernel.

**Client:** process.

**Requests:** read, write system calls.

**Mediation:** U/K bit / system call implementation.

**Principal:** user ID.

**Authentication:** kernel keeps track of user ID for each process.

**Authorization:** permission bits & owner uid in each file's inode.

## Example: Web Server

<b>Resource:</b>	Wiki pages
<b>Client:</b>	any computer that speaks HTTP
<b>Server:</b>	web application, maybe written in Python
<b>Requests:</b>	read/write wiki pages
<b>Mediation:</b>	server stores data on local disk, accepts only HTTP reqs
<b>Principal:</b>	username
<b>Authentication:</b>	password
<b>Authorization:</b>	list of usernames that can read/write each wiki page



## Example: Firewall

**Resource:** internal servers

**Client:** any computer sending packets

**Server:** the entire internal network

**Requests:** packets

**Mediation:**

- internal network must not be connected to internet in other ways
- no open wifi access points on internal network for adversary to use
- no internal computers that might be under control of adversary

**Principal, authentication:** none

**Authorization:** check for IP address & port in table of allowed connections



## What Can Go Wrong?

1. Bypass complete mediation by software bugs
2. Bypass complete mediation by an adversary
3. Policy vs. mechanism
4. Interactions between layers, components
5. Users make mistakes
6. Cost of security

# Bypassing Complete Mediation

## All ways to access resource must be checked by guard

- Common estimate: one bug per 1,000 lines of code
- Adversary may trick server code to do something unintended, bypass guard

## How to prevent bypassing?

- Reduce complexity: reduce lines of code
- The "**principle of least privilege**"

## Example: Paymaxx.com (2005)

**`https://my.paymaxx.com/`**

- Requires username and password
- If you authenticate, provides menu of options
- One option is to get a PDF of the tax form

**`https://my.paymaxx.com/get-w2.cgi?id=1234`**

- Gets a PDF of W2 tax form for ID 1234

**`get-w2.cgi` forgot to check authorization**

- Attacker manually constructs URLs to fetch all data

## Example: SQL Injection

username	email	public?
yubin	yubin@sjtu.edu.cn	yes
mike	mcarbin@sjtu.edu.cn	yes
katrina	lacurts@sjtu.edu.cn	no

```
SELECT username, email FROM users  
WHERE
```

```
username='<username>' AND  
Let <username> = 'katrina' OR username='  
public='yes'
```

```
SELECT username, email FROM users  
WHERE
```

```
username='katrina' OR username=''  
AND
```

# SQL Injection



ZU 0666',0,0);; DROP DATABASE TABLICE;



**AUTHENTICATION: PASSWORD**



# Authentication: Who Are You?

## Something you **know**

- E.g., passwords, answers of questions

## Something you **have**

- E.g., keys or smart cards

## Something you **are**

- E.g., fingerprint, iris, or other biometric method

**Q:** What are their pros and cons?

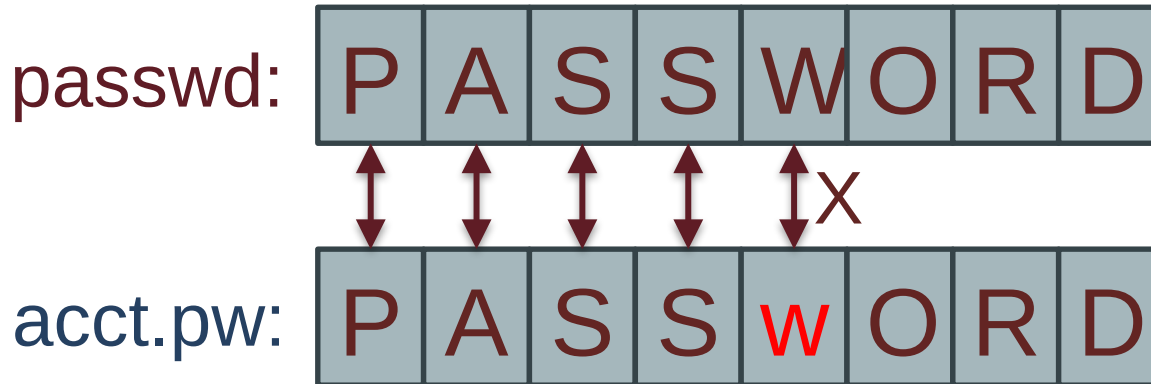
# Authentication: Password

## Password goals

- Authenticate user
- Adversary must guess
  - For random 8-letter passwords,  $\sim 26^8$ ?
- Guessing is expensive

## An Example: Guessing Password (Tenex)

```
checkpw (user, passwd):  
    acct = accounts[user]  
    for i in range(0, len(acct.pw)):  
        if acct.pw[i] ≠ passwd[i]:  
            return False  
    return True
```



## **Problem: Guess One Character at a Time**

### **Guess a password**

- Allocate password 1 byte away from a page boundary
- Page-fault means first char is OK
- Example of cross-layer interactions

### **Problem: server has a copy of all passwords**

- If adversary exploits buffer overflow, can get a copy of all passwords

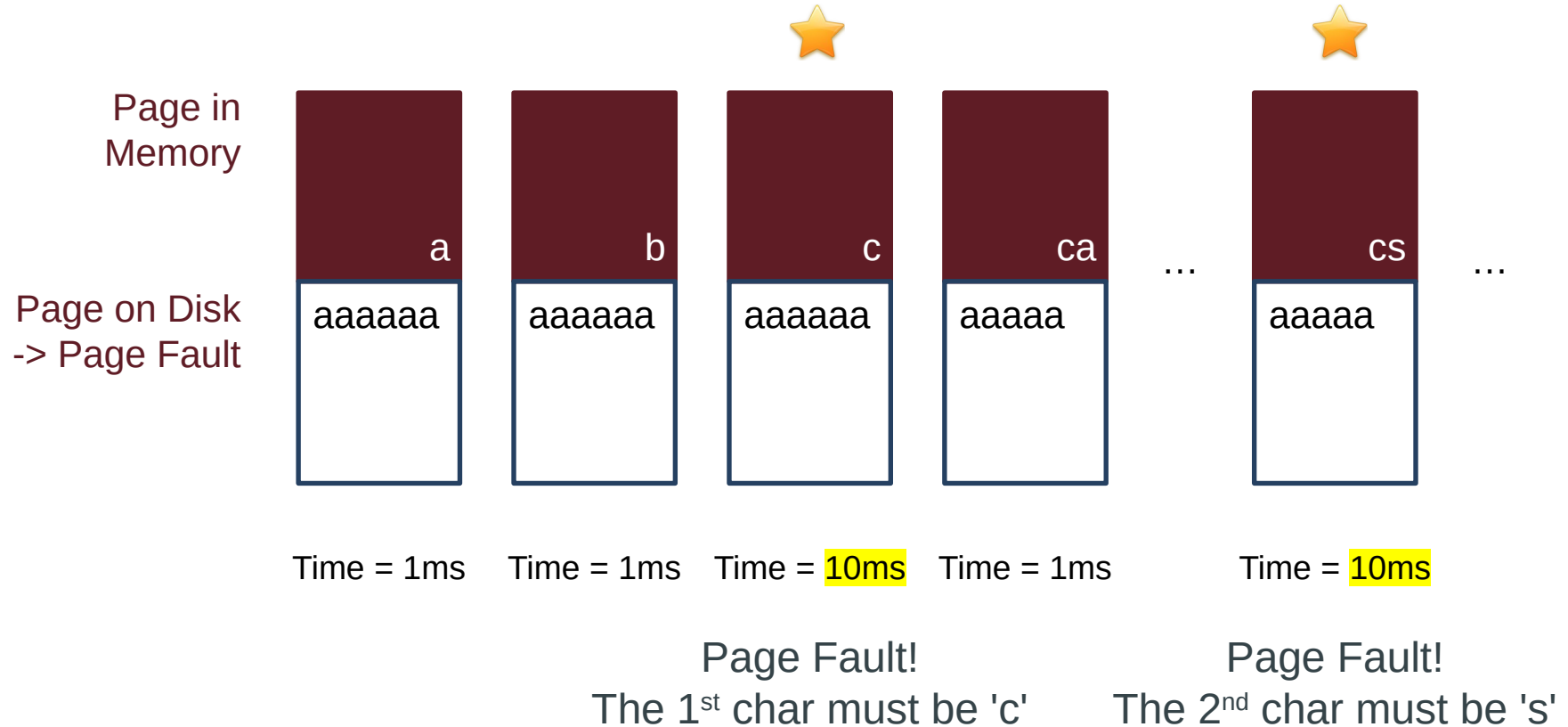
# Timing Attack: Guess One Character at a Time

## Guess a password

- Allocate password 1 byte away from a page boundary
- Page-fault means first char is OK
- Example of cross-layer interactions

Also known as "*Timing Attack*"

# Timing Attack: Guess One Character at a Time



## Idea: Store Hash of Password

### Hash functions: arbitrary strings to fixed-length output

- Common output sizes are 160 bits, 256 bits, ..
- One-way: given  $H(x)$ , hard to recover  $x$
- Collision-resistant: hard to find different  $x$  and  $y$  such that  $H(x)=H(y)$
- Evolve over time: SHA-0 and MD-5 used to be popular, now considered broken

## Store Hash of Password

Accounts database stores hash of every user's password

Compute hash, check if the hash matches

Solves not only the password theft problem, but also the page-fault attack



## Store Hash of Password

```
username | hash(password)
arya      |
de5aba604c340e1965bb27d7a4c4ba03f4798ac7
jon       |
321196d4a6ff137202191489895e58c29475ccab
Sansa     |
check_password(username, inputted_password):
    stored_hash = accounts_table[username]
    inputted_hash = hash(inputted_password)
    return stored_hash == inputted_hash
```

## Store Hash of Password

**What happens if an adversary breaks into a popular web site with 1M accounts?**

- Adversary steals 1M hashes
- Problem: adversary can guess each password one-by-one
- Problem: worse yet, adversary can build table of hashes for common passwords
- Often called a "rainbow table"
- Users are not great at choosing passwords, many common choices

# Password Statistics (32M Password, 2009)

Password Popularity – Top 20

Rank	Password	Number of Users with Password (absolute)
1	123456	290731
2	12345	79078
3	123456789	76790
4	Password	61958
5	iloveyou	51622
6	princess	35231
7	rockyou	22588
8	1234567	21726
9	12345678	20553
10	abc123	17542

Rank	Password	Number of Users with Password (absolute)
11	Nicole	17168
12	Daniel	16409
13	babygirl	16094
14	monkey	15294
15	Jessica	15162
16	Lovely	14950
17	michael	14898
18	Ashley	14329
19	654321	13984
20	Qwerty	13856

5,000 unique password account for 6.4M users (20%)

Similar statistics confirmed again in 2010 (Gawker break-in)

"Consumer Passwords Worst Practices" report by Imperva<sup>47</sup>

# 30 Most Used Passwords in 2022



1	123456	11	abc123	21	princess
2	password	12	1234	22	letmein
3	123456789	13	password1	23	654321
4	12345	14	iloveyou	24	monkey
5	12345678	15	1q2w3e4r	25	27653
6	qwerty	16	000000	26	1qaz2wsx
7	1234567	17	qwerty123	27	123321
8	111111	18	zaq12wsx	28	qwertyuiop
9	1234567890	19	dragon	29	superman
10	123123	20	sunshine	30	asdfghjkl

# Store Hash of Password

## Important to think of human factors when designing security systems

- Not much we can do for users that chose "123456"
- Plenty of bad passwords have numbers, uppercase, lowercase, etc.
- What matters is the (un)popularity of a password, not letters/symbols
- But we can still make it a bit harder to guess less-embarrassing passwords

# Salting

## **Salting: make the same password have different hash values**

- Makes it harder to build a pre-defined lookup table

## **Choose random salt value when storing the password (& store the salt)**

- Store hash of salt and password together
- Use the original salt to compute a matching hash when verifying password
- Every password has many possible hash values -> impractical to build rainbow table

## **Use a much more expensive hash function**

- For reference: look up "bcrypt" by Provos and Mazieres

# Salting

```
username | salt          | hash(password | salt)
arya     | 5334900209 |
c5d2a9ffd6052a27e6183d60321c44c58c3c26cc
jon      | 1128628774 |
624f0ffa577011e5704bdf0760435c6ca69336db
Sansa    | 8188708254 |
5ee2b8e1ce270180e1d14c7d458b1ed9300ce5
Hodor    | 6209415273 | accounts_table[username]
f7e17e61376116ca23360915b578d923d86e0319

inputted_hash = hash(inputted_password |
                        salt)

return stored_hash == inputted_hash
```

# Bootstrap Authentication

**Do not want to continuously authenticate with password for every command**

- Typing, storing, transmitting, checking password: risk of compromise

**In Unix login:**

- Login process exchanges password for **userid**

**In web applications:**

- Often exchange password for a session **cookie**
- Can think of it as a temporary password that's good for limited time



## Session Cookies: Strawman

First check username and password, if ok, send:

```
{username, expiration, H(server_key | username |  
expiration)}
```

**Use the tuple to authenticate user for a period of time**

- Nice property: no need to store password in memory, or re-enter it often
- *Server\_key* is there to ensure users can't fabricate hash themselves
- Arbitrary secret string on server, can be changed (invalidating cookies)
- Can verify that the username and expiration time is valid by checking hash

## Session Cookies: Strawman

**Problem: the same hash can be used for different username/expiration pairs!**

- E.g., "Ben" and "22-May-2012" may also be "Ben2" and "2-May-2012"
- Concatenated string used to compute the hash is same in both cases!
- Can impersonate someone with a similar username

**Principle: be explicit and unambiguous when it comes to security**

- E.g., use an invertible delimiter scheme for hashing several parts together

# Phishing Attacks

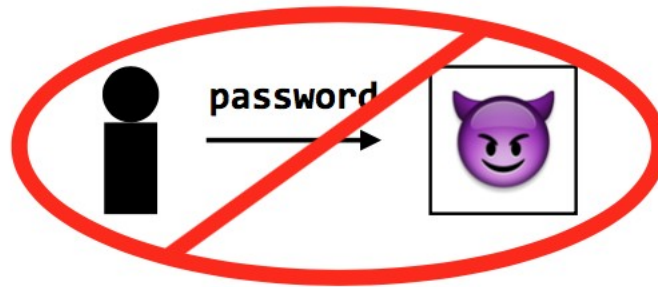
**Adversary tricks user into visiting legitimate-looking web site (e.g., bank)**

- Asks for username/passwd
- Actually a different server operated by adversary
  - E.g., **bank0famerica.com** instead of **bankofamerica.com**
- Stores any username/passwd entered by victims
- Adversary can now impersonate victims on the real web site

## Key Problem of Password

**Key problem:** once you send a password to the server, it can impersonate you

- Solved part of the problem by hashing the password database
- But still sending the password to the server to verify on login

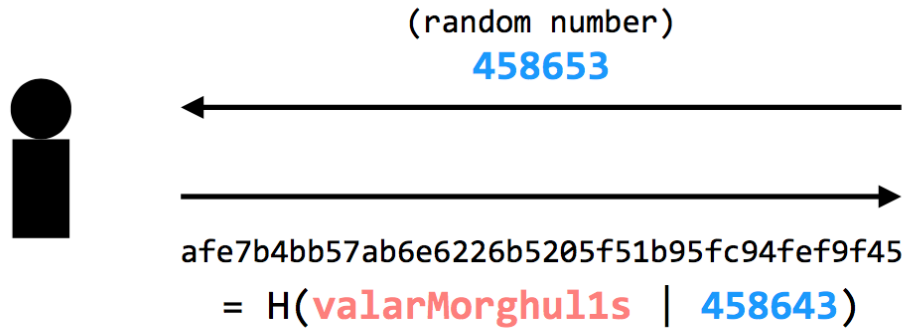


# Tech 1: challenge-response scheme

## Challenge-response

- Server chooses a random value **R**, sends it to client
- Client computes **H(R + password)** and sends to server
- Server checks if this matches its computation of hash with expected password
- If the server didn't already know password, still doesn't know

# Tech 1: challenge-response scheme



**password is never sent directly**

**valid server**

username	password
arya	<b>valarMorghul1s</b>
jon	w1nterIsC0ming
sansa	LemonCakesForever
hodor	hodor

server computes  
H(**valarMorghul1s** | **458643**)  
and checks

Adversary only learns H(valarMorghul1s | 458643); can not recover the password from that

## Tech 2: use passwords to authenticate the server

### Make the server prove it knows your password

- Client chooses  $Q$ , sends to server, server computes  $H(Q + \text{password})$ , replies
- Only the authentic server would know your password!

### Unfortunately, not many systems use this in practice

- In part because app developers just care about app authenticating user...

## Tech 2: use passwords to authenticate the server

**Complication: could be used with first scheme to fool server!**

- First, try to log into the server: get the server's challenge **R**
- Then, decide you want to test the server: send it the same **R**
- Send server's reply back to it as response to the original challenge

**Principle: be explicit, again!**

- E.g., hash the intended recipient of response (e.g., client or server), and have the recipient verify it



## Tech 3: turn offline into online attack

### Turn phishing attacks from offline into online attacks

- If adversary doesn't have the right image, users will know the site is fake
- Adversary could talk to real site, fetch image for each user that logs in


# "Sitekey"

Home Banking x +

← → ↻ <https://global1.onlinebank.com/vtg/preauthenticaterauser.htm?loginUrl=4D4E35D99687A9327764> 🔒 ☆ 🔧

[Print](#) | [Help](#)

**Log In to Your Account**




not an elephant

Please Enter Your Password Only After Verifying Your Personal Image.

User ID : zeldovich1

Password :

[Forgot your Password?](#)

 [Back](#) [Login](#)

## Tech 3: turn offline into online attack

### Why is it still useful, then?

- Requires more efforts on adversary's part to mount attack
- Even if adversary does this, bank can detect it
- Watch for many requests coming from a single computer
- That computer might be trying to impersonate site
- Turns an offline/passive attack into an online/active attack

### Key insight

- Don't need perfect security, small improvements can help

## Tech 4: Specific password

### **Make passwords specific to a site**

- Instead of sending password, send  $H(\text{servername} + \text{password})$
- Just like a basic password scheme, from the server's point of view

**Except impersonator on another server gets diff passwd**

### **Recommendation**

- E.g., LastPass

## Tech 5: one-time passwords

### One-time Password

- If adversary intercepts password, can keep using it over and over
- Can implement one-time passwords: need to use a different password every time

### Design: construct a long chain of hashes.

- Start with password and salt, as before
- Repeatedly apply hash,  $n$  times, to get  $n$  passwords
- Server stores  $x = H(H(H(H(\dots(H(\text{salt}+\text{password})))))) = H^n(\text{salt}+\text{password})$

### To authenticate, send $\text{token} = H^{\{n-1\}}(\text{salt}+\text{password})$

- Server verifies that  $x = H(\text{token})$ , then sets  $x \leftarrow \text{token}$
- User carries a printout of a few hashes, or uses smartphone to compute them.

## Tech 5: one-time passwords

**Alternative design: include time in the hash (Google's 2-step verification)**

- Server and user's smartphone share some secret string  $K$
- To authenticate, smartphone computes  $H(K \parallel \text{current time})$
- User sends hash value to server, server can check a few recent time values.

# Google's App-specific password

## Application-specific passwords

### Step 2 of 2: Enter the generated application-specific password

You may now enter your new application-specific password into your application.  
Note that this password grants complete access to your Google Account. For security reasons, it will not be displayed again:

**fmin nnwg bftf dppi**

No need to memorize this password.  
You should need to enter it only once. Spaces don't matter.

Done

Reeder on MBA  
Chrome on MBA  
Gmail on iPhone  
Test for CSE

Dec 2, 2012  
Dec 2, 2012  
Dec 10, 2012  
Dec 20, 2012

Dec 19, 2012  
Dec 2, 2012  
Unavailable  
Unavailable

[ [Revoke](#) ]  
[ [Revoke](#) ]  
[ [Revoke](#) ]  
[ [Revoke](#) ]

## Tech 6: bind authentication and request authorization

**One way to look at problem: sending password authorizes any request**

- Even requests by adversary that intercepts our password
- A different design: use password to authenticate any request
- `req = { username, "write XX to exam.txt", H(password + "write ..") }`

**Server can check if this is a legitimate req from user using password**

- Even if adversary intercepts request, cannot steal/misuse password
- In practice, don't want to use password, use some session token instead
- Could combine well with one-time passwords



# Tech 7: FIDO: Replace the Password

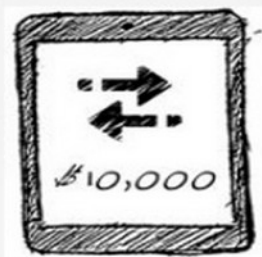


ONLINE AUTH REQUEST

LOCAL DEVICE AUTH

SUCCESS

## PASSWORDLESS EXPERIENCE (UAF standards)



Transaction Detail



Show a biometric



Done

## SECOND FACTOR EXPERIENCE (U2F standards)



Login & Password



Insert Dongle, Press button



Done

fido

[Store](#)[Mac](#)[iPad](#)[iPhone](#)[Watch](#)[AirPods](#)[TV & Home](#)[Only on Apple](#)[Accessories](#)[Support](#)

## Newsroom

[Search Newsroom](#)[Popular Topics](#)

PRESS RELEASE

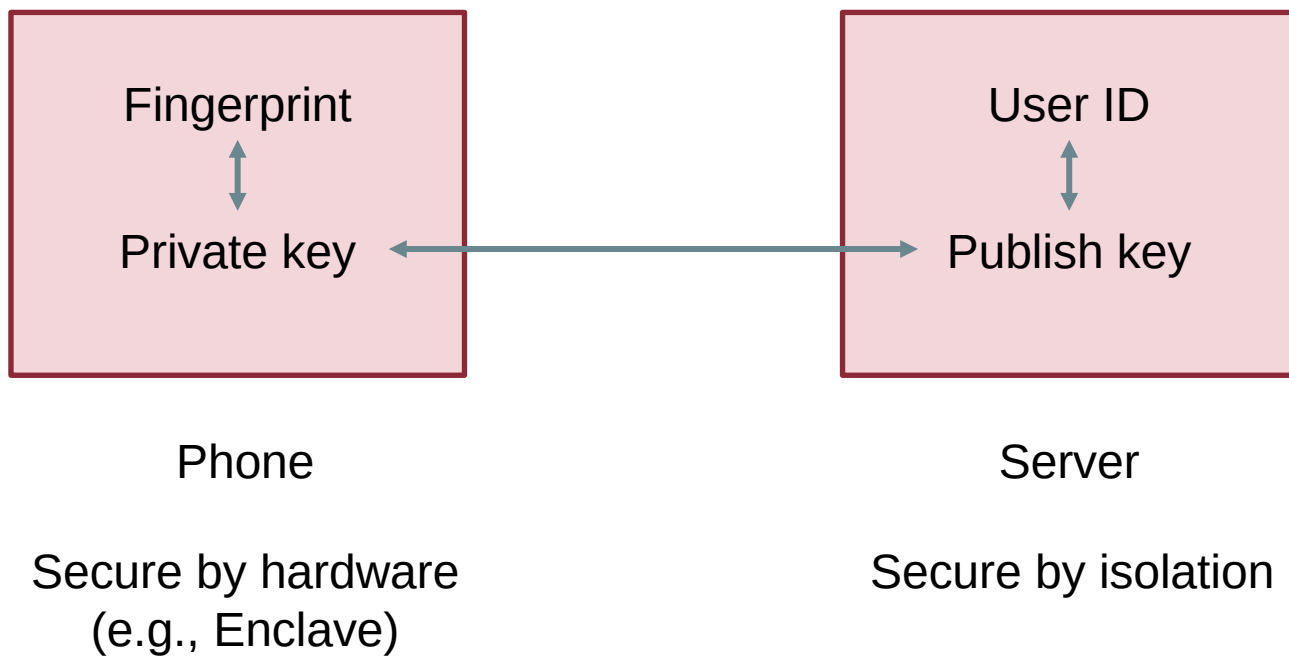
May 5, 2022

# Apple, Google, and Microsoft commit to expanded support for FIDO standard to accelerate availability of passwordless sign-ins

Faster, easier, and more secure sign-ins will be available to consumers across leading devices and platforms



## Three Bindings



# Bootstrapping

## How to initially set a password for an account?

- If adversary can subvert this process, cannot rely on much else
- MIT: admissions office vets each student, hands out account codes
- Many web sites: anyone with an email can create a new account

## Changing password (e.g., after compromise)

- Need some additional mechanism to differentiate user vs. attacker
- MIT: walk over to accounts office, show ID, admin can reset password
- Many sites: additional "security" questions used to reset password

## Password bootstrap / reset mechanisms are part of the security system

- Can sometimes be weaker than the password mechanisms
- Sarah Palin's Yahoo account was compromised by guessing security Q's
- Personal information can be easy to find online



# SECURITY PRINCIPLES

# Principle of Least Privilege

## Deal with bugs / incomplete mediation

- Any component that can arbitrarily access a resource must invoke the guard
- If component has a bug or design mistake, can lead to incomplete mediation
- General plan: reduce the number of components that must invoke the guard
  - E.g., arrange for DB server to check permissions on records returned, then security does not depend as much on *lookup.cgi*

# Least Trust

Privileged components are "trusted"

- **Trusted is bad**: you are in trouble if a trusted component breaks
- **Untrusted components are good**: does not matter if they break
- Good design has **few trusted components**, other things do not affect security

# Users Make Mistakes

## Social engineering, phishing attacks

- How many of you really read the permission acquired by apps during installing?

**Good idea: threat model should not assume users are perfect**



# Cost of Security

## Security VS. Availability

- E.g., system requires frequent password changes -> users may write them down

## How far should a teacher go to protect the exam file?

- Put the file on a separate computer, to avoid sharing a file system?
- Disconnect computer from the network, to avoid remotely exploitable OS bugs?
- Put the server into a separate machine room?
- Get a guard to physically protect the machine room?
- ...

**Good idea: cost of security mechanism should be commensurate with value**

BO55man69

