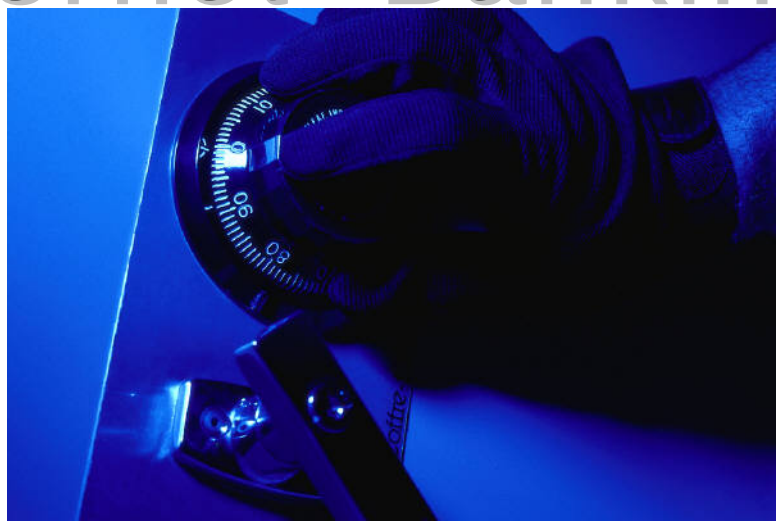
FMA-RMS

Hacking Internet Banking Applications



HackInTheBox 2005
Kuala Lumpur

By Fabrice A. Marie
fabrice.marie@fma-rms.com

Goal of this presentation

1. Showcase terms well known internet banking applications attacks…
2. … and of course less well known internet attacks.
3. Explain basically the best case (for the attacker) scenarios of the attacks.
4. Give guidelines for procurement of banking applications.
5. Give guidelines for additional contractual requirements to take into account.

Table of Contents

1. Introduction

1.1. Banking application architecture

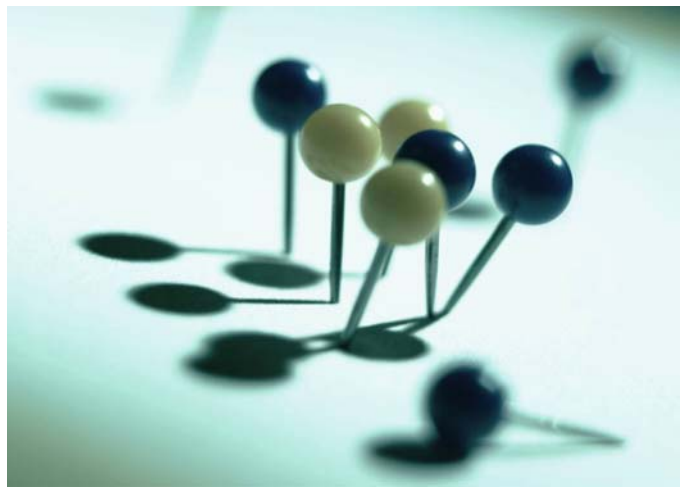
1.2. Common Technologies

1.3. What is usually good

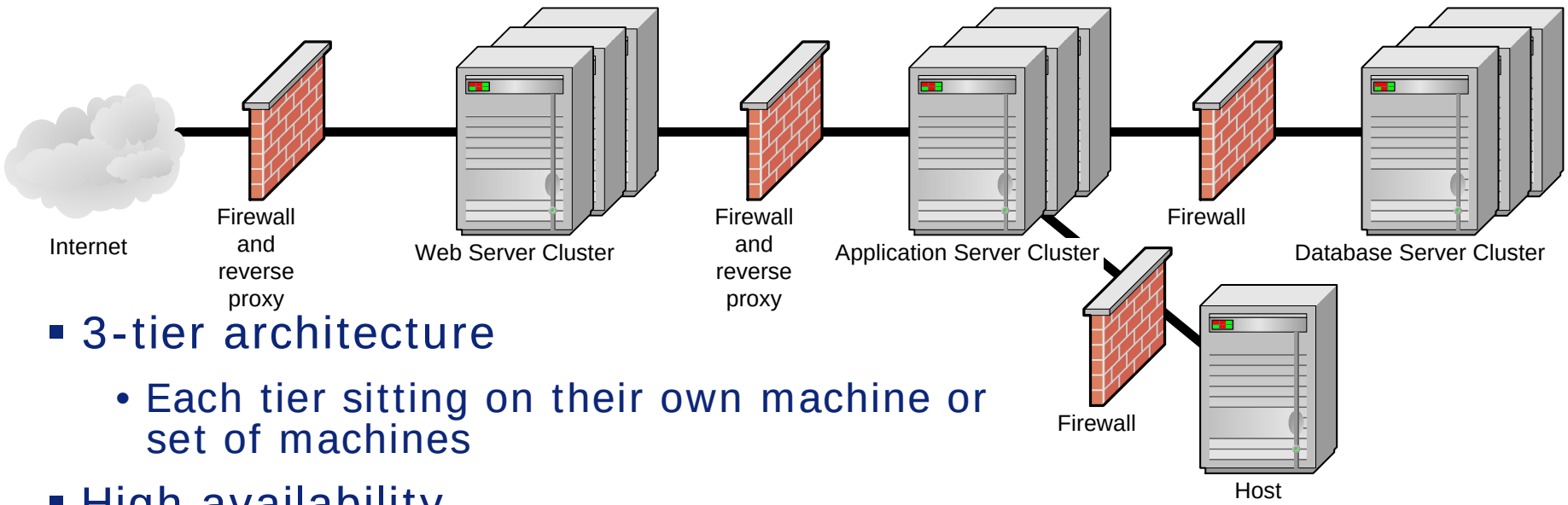
1.4. What is usually NOT good

1.5. Requirements for successful attack

1.6. Tools used



Banking Application Architecture



- 3-tier architecture
  - Each tier sitting on their own machine or set of machines
- High availability
  - Each tier clustered to ensure fault tolerance and HA
- Firewalls
  - Each tier fronted by a firewall
- Reverse proxy
  - Reverse proxies in front of the web server and the application server

Common Technologies

- Java J2EE
  - WebLogic from BEA
  - WebSphere from IBM
  - Sun One Application Server from Sun
  - Pros and Cons:
    - Development done in Java exclusively.
    - Platform independent.
    - Application servers are certified J2EE.
    - Unfortunately all of them offer their unique “better” features.
      - These “better” unique features are not standard.
      - Not easy to migrate from one J2EE app server to another.

Common Technologies - continued

- Microsoft .Net
  - Pros and Cons
    - Development done in a myriad of .Net Languages.
      - Very simple to shoot yourself in the foot.
      - Documentation widely available but somewhat hard to search through.
    - The whole solution supporting the application developed by one consistent software provider: Microsoft
    - .NET application run almost exclusively on Microsoft platform.
      - Mono (**http://www.mono-project.com/**) is working, but no financial application developed for it yet.

What is Usually Good

- Network and system architecture
- Session Management
- Password RSA encrypted from end-to-end
  (on top of SSL, it is a requirement in Singapore at least)
- Good quality audit trails

→ That's all !

What is usually **NOT** good

- Lack of input validation
  - Failure to check that the parameter even exists.
  - Failure to ensure the parameters is really what the application expects.
- Bad quality code
  - Failure to check return code or fails to catch exception.
  - Failure to ensure that the pointer is not null before dereferencing it.
- Programmers become more lazy
  - They create generic framework that cannot adapt properly to a particular exceptional case.
  - They use the wrong “hammer” to solve a particular problem.
  - They are reluctant to log in details every exceptional behavior.

What is usually **NOT** good - continued

[Seriously, I'm not joking…]

- Web programmers believe that a hidden field is really hidden.
- Web programmers believe that a drop down select box cannot have any other value than the one displayed in the browser
- Web programmers believe that when the menu does not show a functionality it means you cannot run it
- Web programmers believe that cookies are different from any other input
- Web programmers believe that hexadecimal or base64 encoding is some form of encryption

→ And much more !

Requirements for a Successful Attack

- Need an account with the bank
- A valid username and password
- No smartcard based end-to-end SSL

  (Proxy would have to be modified to read certs/keys from Smartcard)

→ That's all !

- Some attacks still possible without these conditions of course.
  - They are just harder to carry out

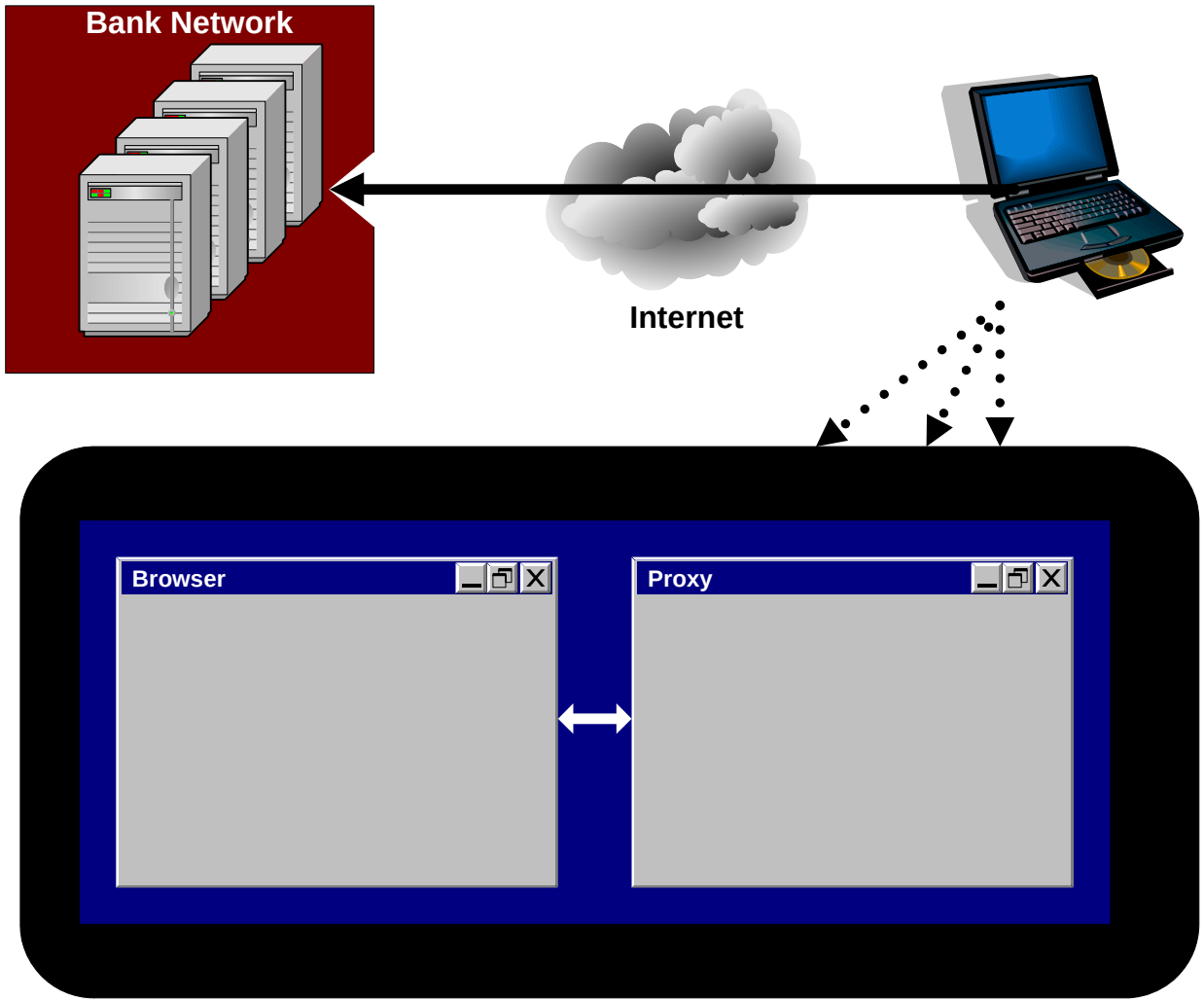
Tools used

- Interactive web proxies
  - Burp (java)
  - Paros (java)
  - Spike (python)
  - Proxomitron (compiled – Windows only)
- Decompilers
  - jad (java)
  - Reflector (.NET)
- Encoders and decoders for
  - MIME, base64, uuencode, hexadecimal

All these tools are free and easy to download from the internet

- Lots of internet applications are broken so we unfortunately have to use IE often

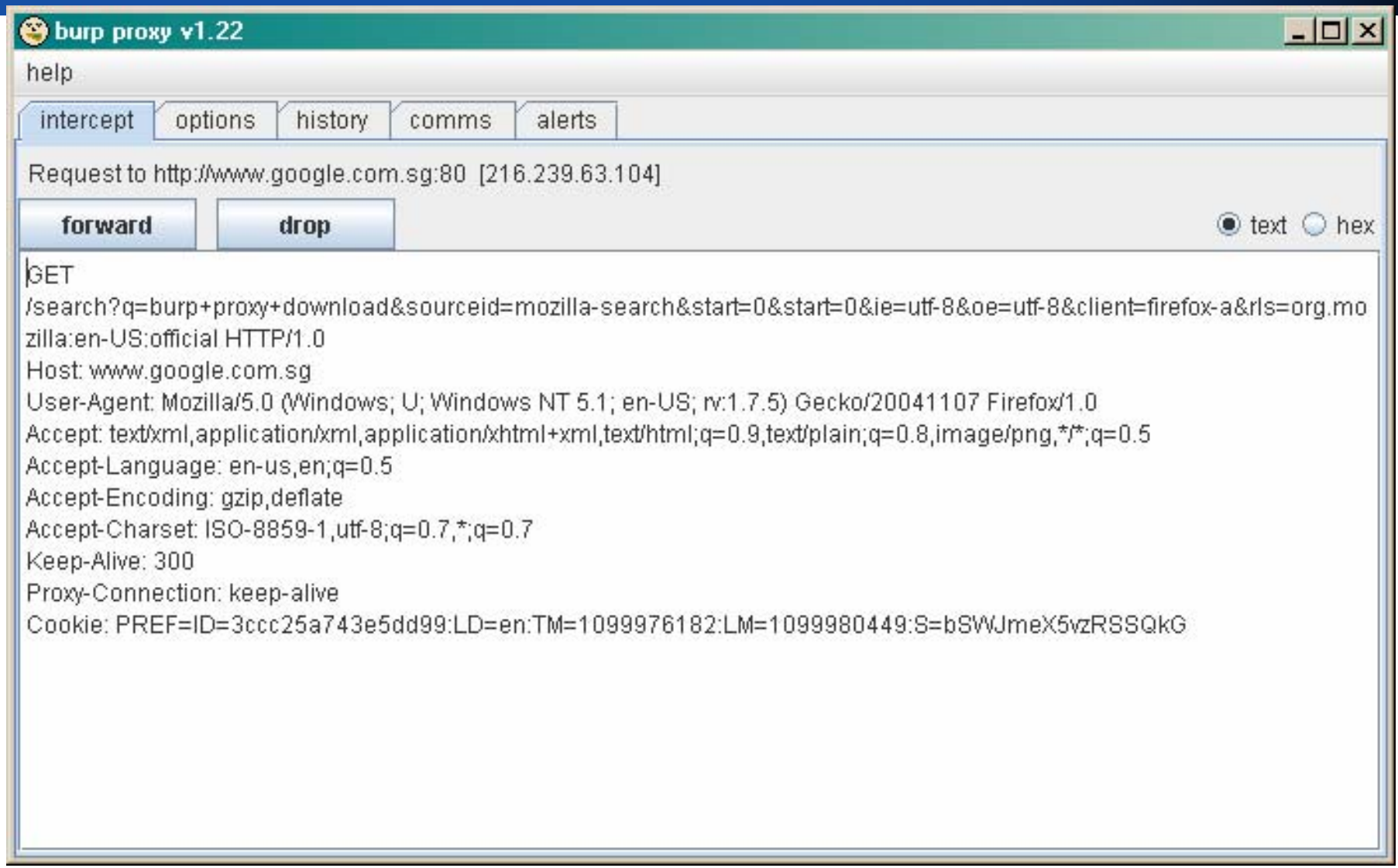
Tools used – Topology of Attack



The proxy helps the attacker to hijack his **own** requests in order to modify them freely.

Goal: bypass client security (JavaScript on the browser)

Tools used – e.g. burp proxy



2. Attacks: Usual Suspects

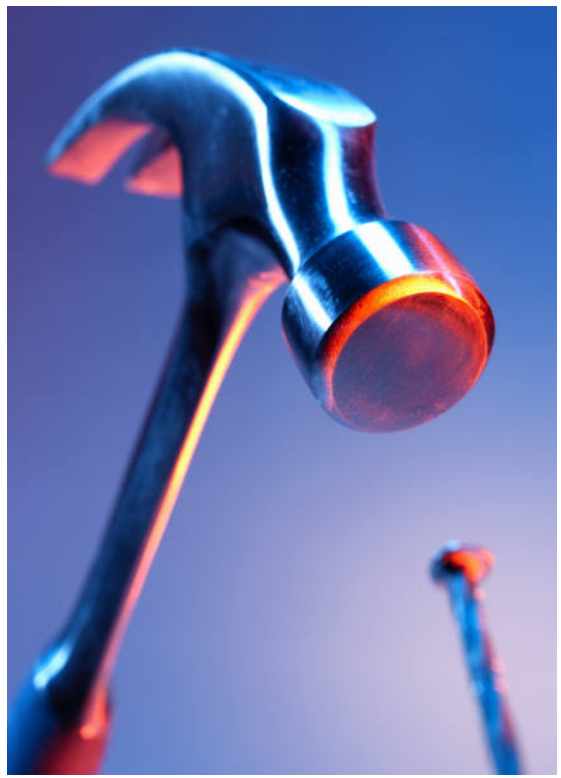
2.1. Cross site scripting

2.2. SQL Injections

2.3. Buffer overflows

2.4. Weak scripts

2.5. Various denials of service



Cross Site Scripting (XSS)

- Generally **high chance of success**
- Worse enemy of an internet banking application
  - Can create scarily accurate fakes.
  - XSS allows the attacker to
    - Steal cookies.
    - Trick the user to give them their credentials
    - Modify the appearance of the page.
    - Execute all sorts of malicious java-script code.
    - **Hard to differentiate for a user** (same SSL cert., same URL, etc…)
- Effective only when no authentication required
  - They are usually on the following pages:
    - “login” or “logout” page.
    - “lost password” / “reset password request” page.
- **For the attack to succeed, the victim still has to be tricked**

SQL Injection

- Generally **small chance of success**
  - Object oriented languages when properly used almost rule out SQL injections.
- Less effective than Cross Site Scripting for stealing money.
- Very interesting to modify or steal personal data.
  - Because it leaves almost no trace
- Mostly found on search pages containing complex options

Buffer Overflow

- Generally **very small chance of success**
  - Object oriented languages when properly used almost rule out buffer overflows.
- Firewall will prevent connect-back most of the times.
- Very interesting to gain access to the internal network.
- Failed buffer overflows often lead to application DoS.
  - When the application is down, the bank will be aware of the attack too fast
- Mostly interesting when found on the login page.

Weak Scripts

[Not all scripts are created equal…!!]

- Some are especially weaker than others:
  - No input validation whatsoever
  - Don’t even check that the parameters are present
  - Verbose error messages
  - Some don’t even compile!
- The weakest are extremely easy to find
  - just call them without any parameter and watch!
  - easy to script once you have a list of URLs
  - sometime they even DoS the whole application

Denial of Service (DoS)

– "It's always easier to destroy than to build…"

- Common Denials of Service:
  - Recursive operation
  - Extremely long timeouts
  - Blocking operations
  - Buffer overflows

Denial of Service (DoS) – continued

Recursive operation

- The script is supposed to call another script
- But attacker tricked it to call itself
  - → use up all the threads of the server
  - → **Extremely effective way to DoS an application**

Extremely long timeouts

- The operation should complete in a short time
- But attacker forced it to take longer
  - → use up all the threads of the server

Denial of Service (DoS) – continued

Blocking operation

- The script is supposed to read a file or a socket and return data
- But attacker tricked it to read a special blocking file

  → use up all the threads of the server

Buffer overflow

- The operation expects a buffer of a certain size
- But attacker overflowed the buffers

  → the thread (or the app server) get killed by the OS.

Corporate Espionage – Loss of Confidentiality

3.1. Spying on competitor's transaction history

3.2. Spying on competitor's bill payments

3.3. Spying on competitor's banking messages

3.4. Spying on VIP's or competitor's credit card bills

Introduction to “read” logic flaws

- A “read” Logic Flaw is present when you trick the application into reading and returning data that it is not supposed to be served to you
  - Example: read other people personal or financial information
- All applications basically perform the requests instructed
- Simply ask the application for the “forbidden” information !
- Most of the time, the application will not double-check and **will serve** you the content requested, **although you are not authorized**

Introduction to “read” logic flaws (cont’d)

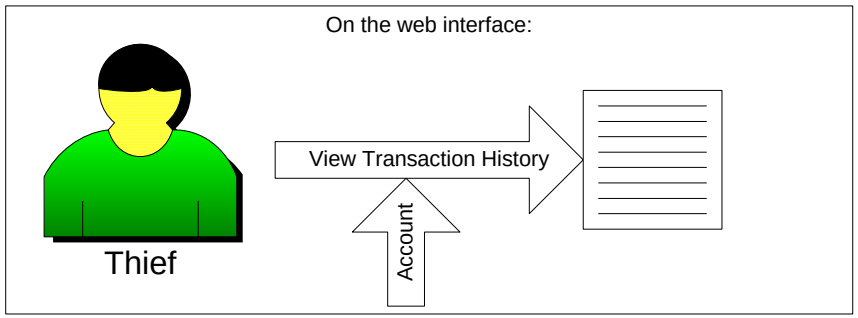
- The application will need some parameters for the requests
  - If you want to see your transaction history you need to feed the application your account number, and begin/end dates.
- Eventually the application always need more than one parameter before it can serve your read request.
- **If** all the parameters for the request are passed to the script from the client’s browser, then **bingo** !!
  - You can modify all the parameters.
  - Therefore you can ask for anything **as long as the application does not check**
    - **APPLICATIONS DO NOT CHECK !!!!**

Introduction to “read” logic flaws (cont’d)

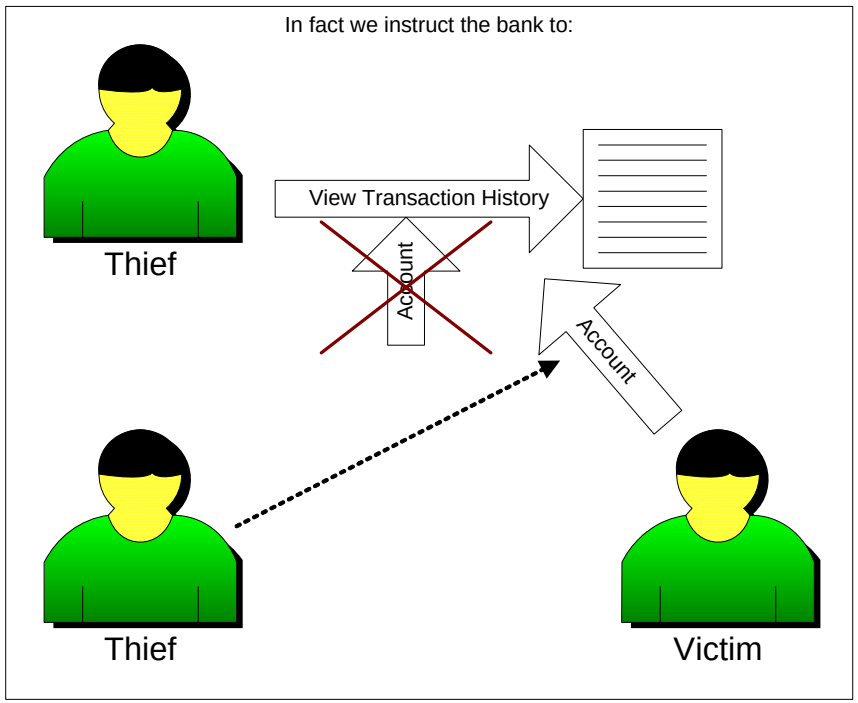
- Some applications are written in more “sane” manner.
- Only parts of the necessary parameters are in the hand of the client’s browser, the rest is kept on the server side attached to the session.

- Much harder to attack in this case
  - **Usually still possible as programmers make other mistakes**

Spying on Competitor's Transaction History

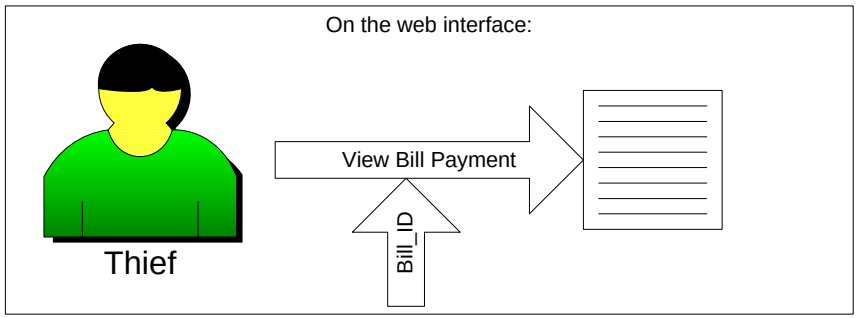


View Transaction History
Of account: Thief

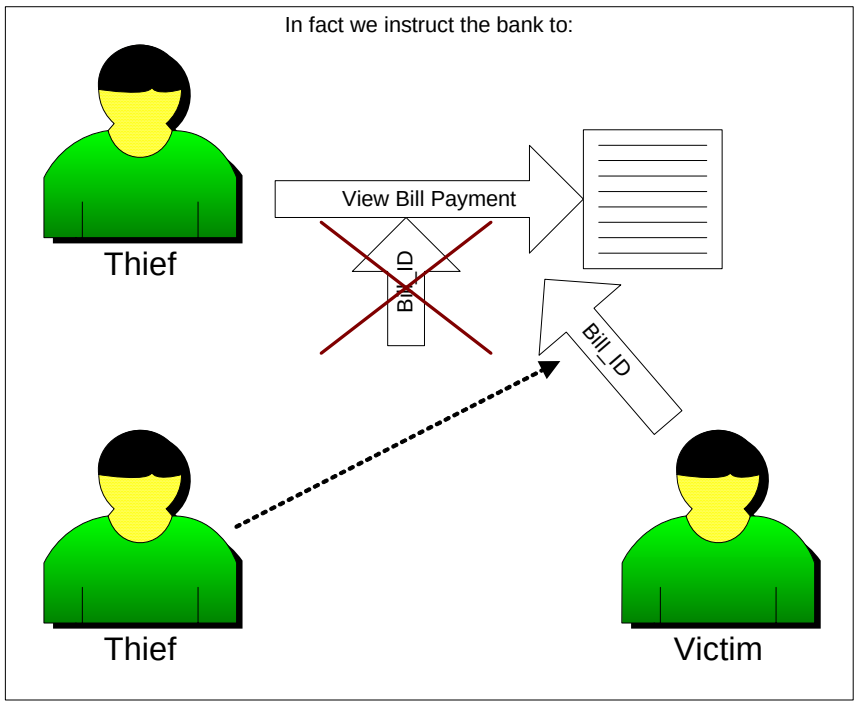


View Transaction History
Of account: **Victim**

Spying on Competitor's Bill Payments

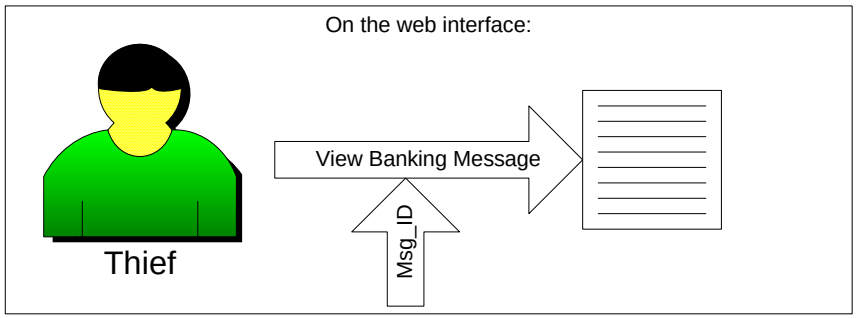


View Bill Payment Info
Of user: Thief
Bill_ID: Thief's bill



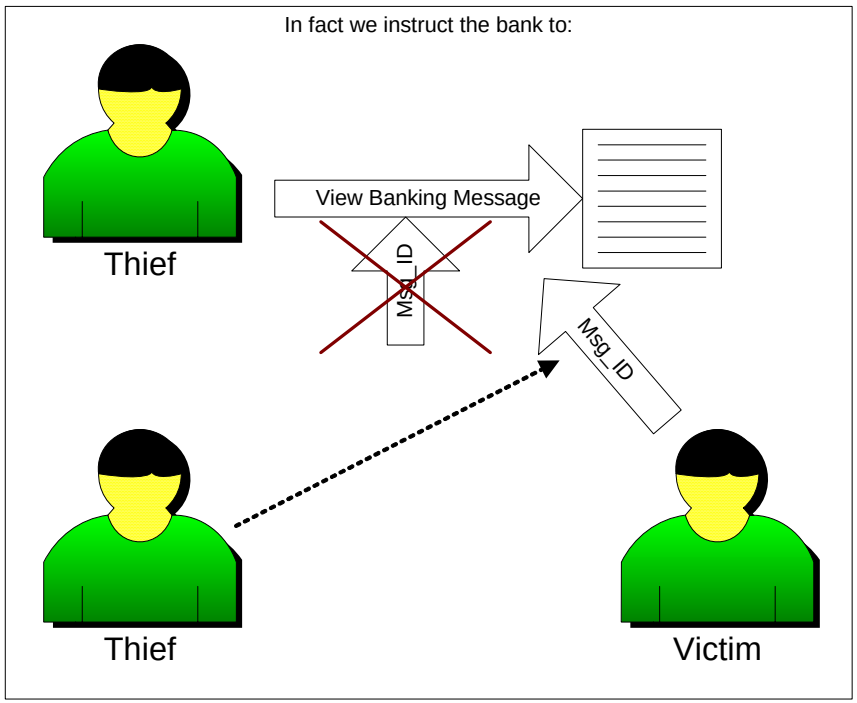
View Bill Payment Info
Of user: **Victim**
Bill_ID: **Victim's bill**

Spying on Competitor's Banking Messages



View Banking Message

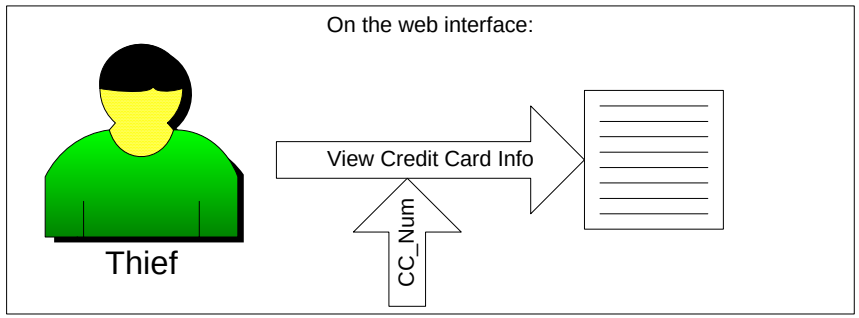
Msg_ID: Thief's message



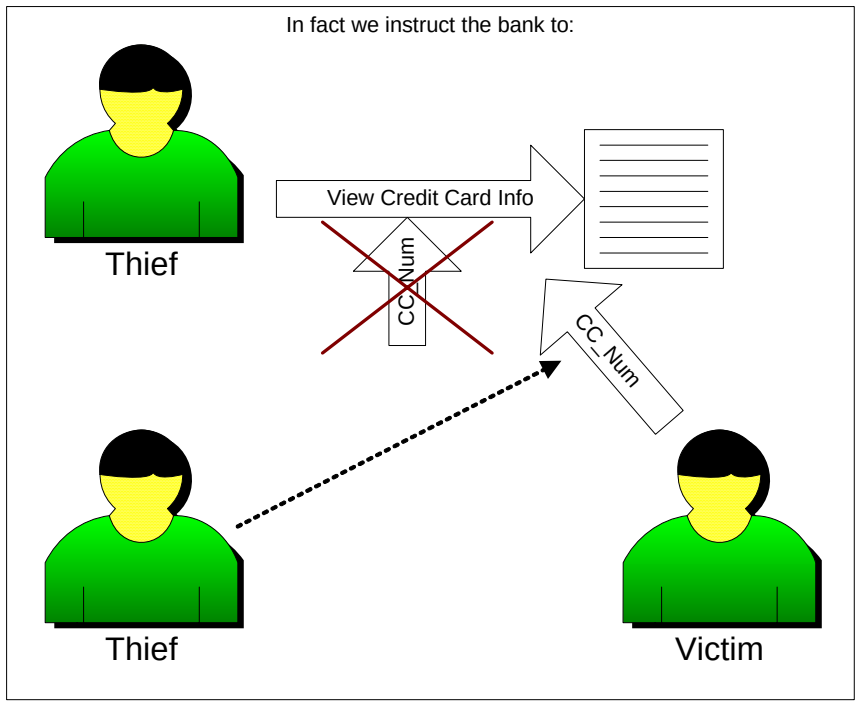
View Banking Message

Msg_ID: **Victim's message**

Spying on VIP or Competitor Credit Card Bills



View Credit Card Info
CC_Num: Thief's card



View Credit Card Info
CC_Num: **Victim's card**

Protection against “read” logic flaws

- Protection is very **straight forward**
- Developers often **forget** to enforce it though

→Keep parameters as much as possible on the server side.

→Pass parameters by reference instead of passing them by value.

→**Verify that the data returned is really owned by the requesting user!**

Interesting Attacks: Outright Frauds

4.1. Stealing money using Fund Transfer functionality.

4.2. Stealing money using Cashier Orders functionality.

4.3. Buying shares at a discounted price.

4.4. Buying shares for free.

4.5. Avoiding various Transaction fees.

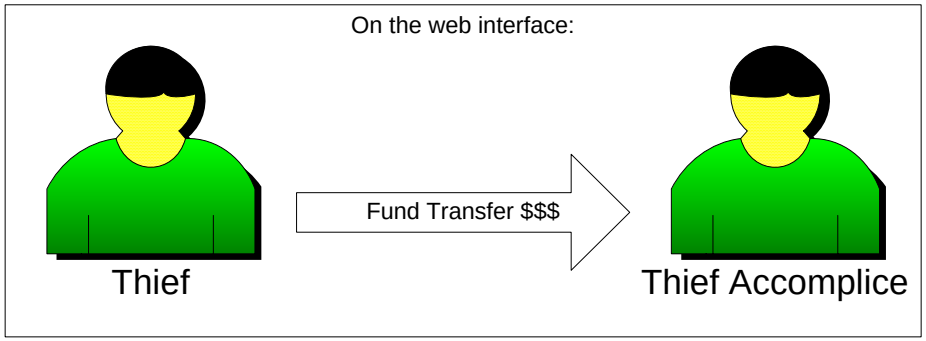
4.6. Purchasing Insurance for free.

4.7. Changing victim's payee information.

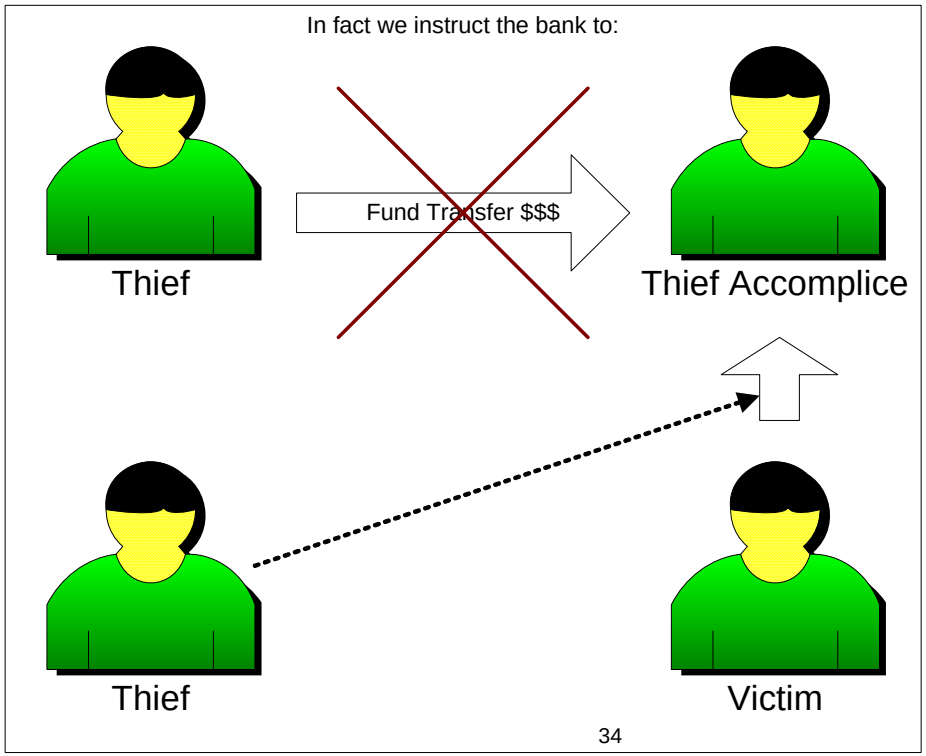
Introduction to “write” logic flaws

- A “write” Logic Flaw is present when you trick the application into performing requests that that have a lasting (“written”) result, and you were not supposed to be authorized.
  - Example: Modify personal information of others, unauthorized money transfers, etc…
- All applications basically perform the requests instructed
- Simply ask the application to perform the fraud!
- Most of the time, the application will not double-check and **will execute the request**, resulting in an **outright fraud**.

Stealing Money Using Fund Transfer Functionality

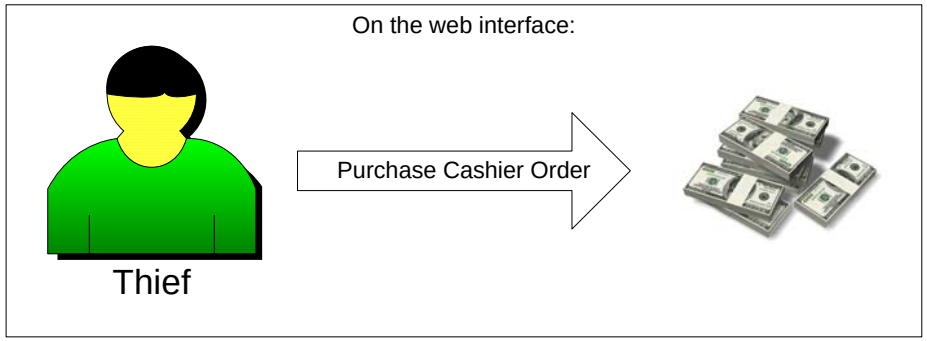


Fund transfer
From: Thief
To: Thief Accomplice

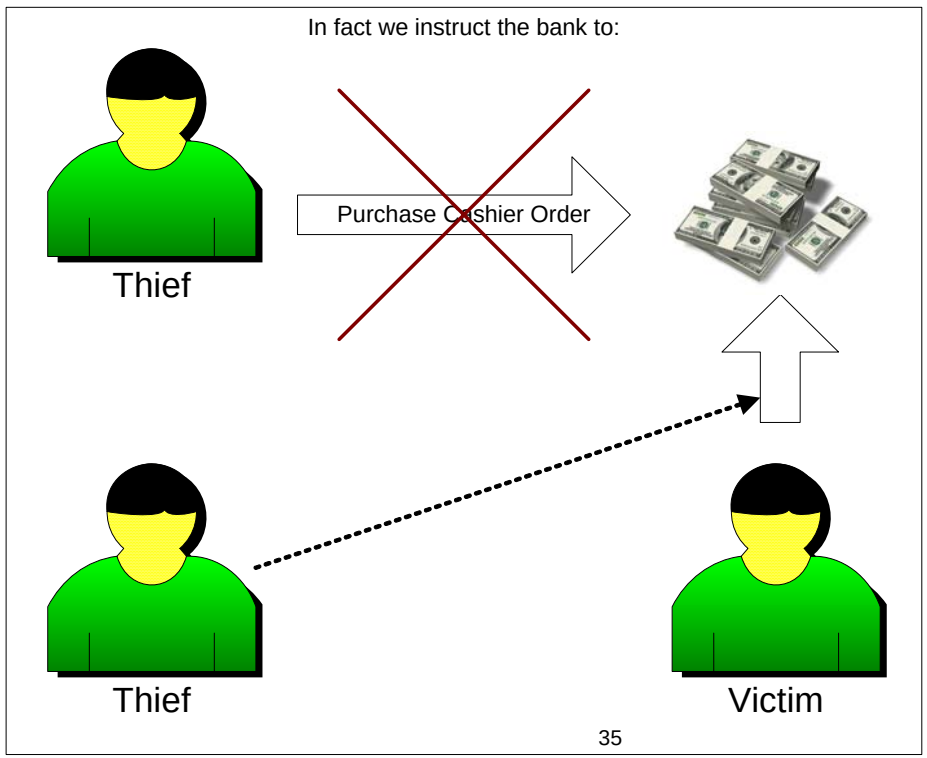


Fund Transfer
From: **Victim**
To: Thief Accomplice

Stealing Money Using Cashier Order Functionality

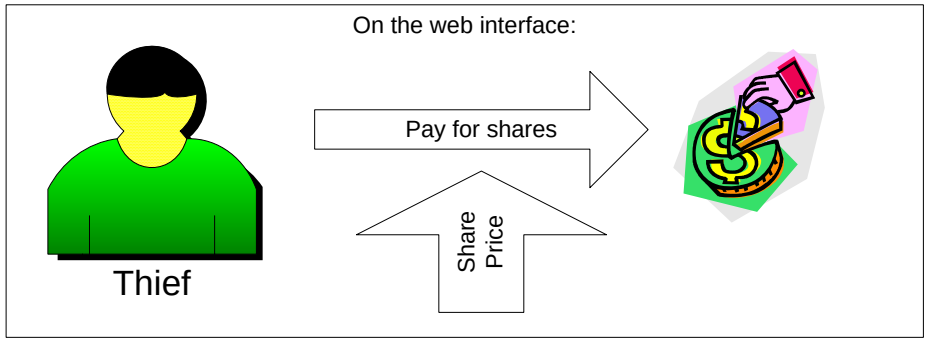


Purchase
Cashier order
For: Thief
From account: Thief

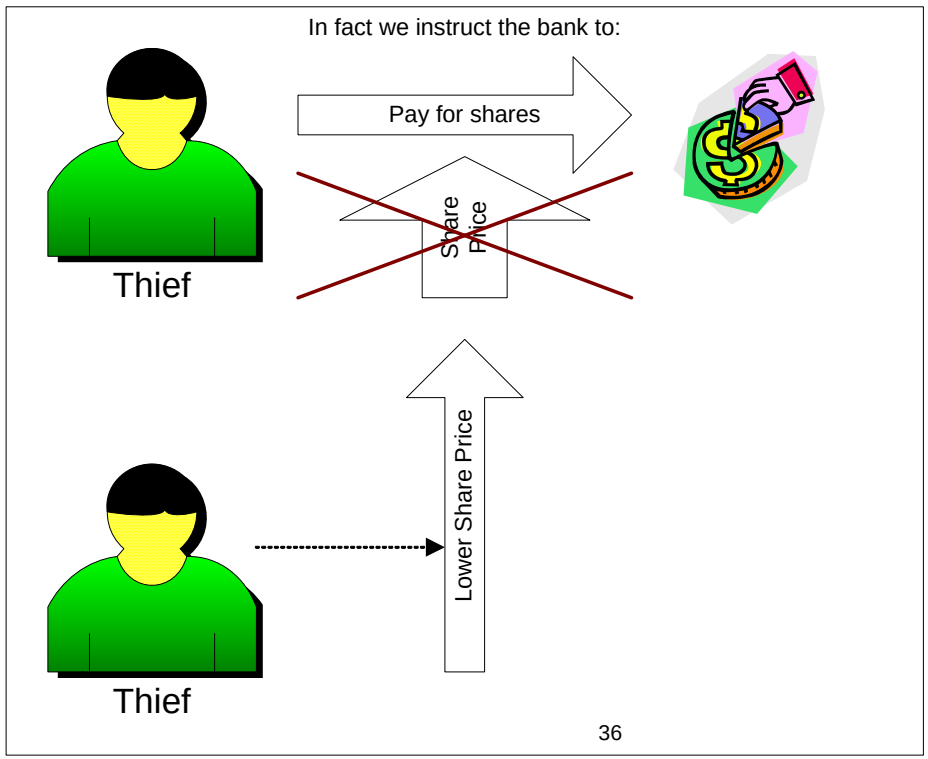


Purchase
Cashier order
For: Thief
From account: **Victim**

Buying Shares at a Discounted Price

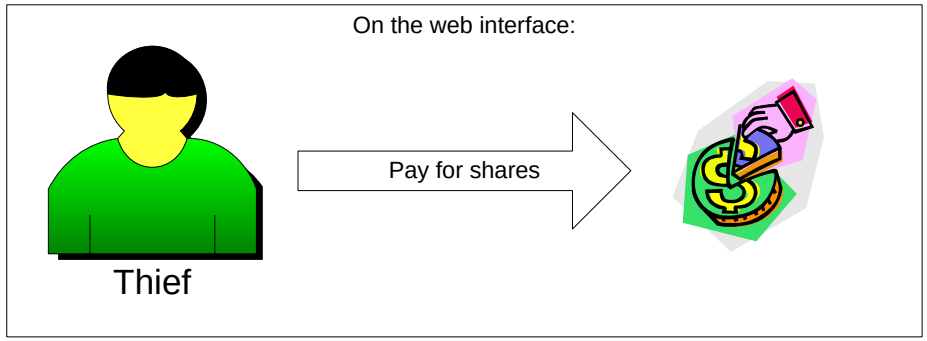


Purchase Shares
Shares for: Thief
Paid by: Thief
Number of units: 100
Price per unit: $10

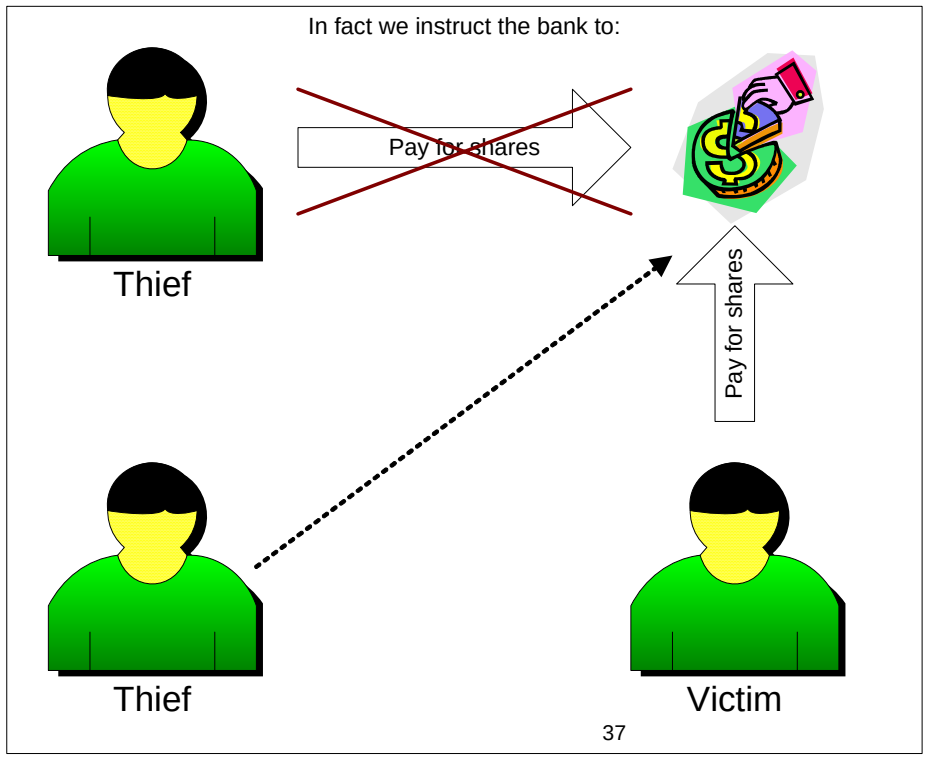


Purchase Shares
Shares for: Thief
Paid by: Thief
Number of units: 100
Price per unit: **$1**

Buying Shares for Free

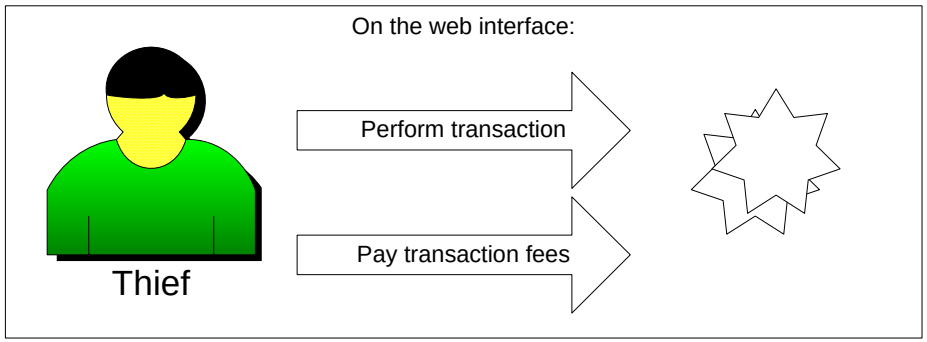


Purchase Shares
Shares for: Thief
Paid by: Thief

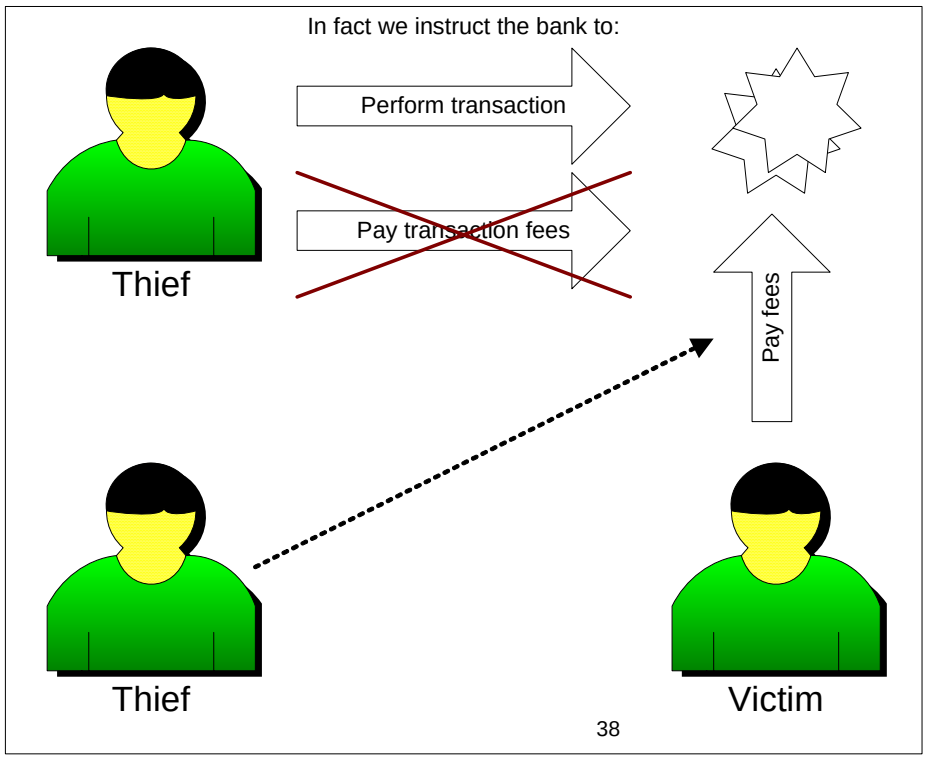


Purchase Shares
Shares for: Thief
Paid by: **Victim**

Avoiding Various Transaction Fees

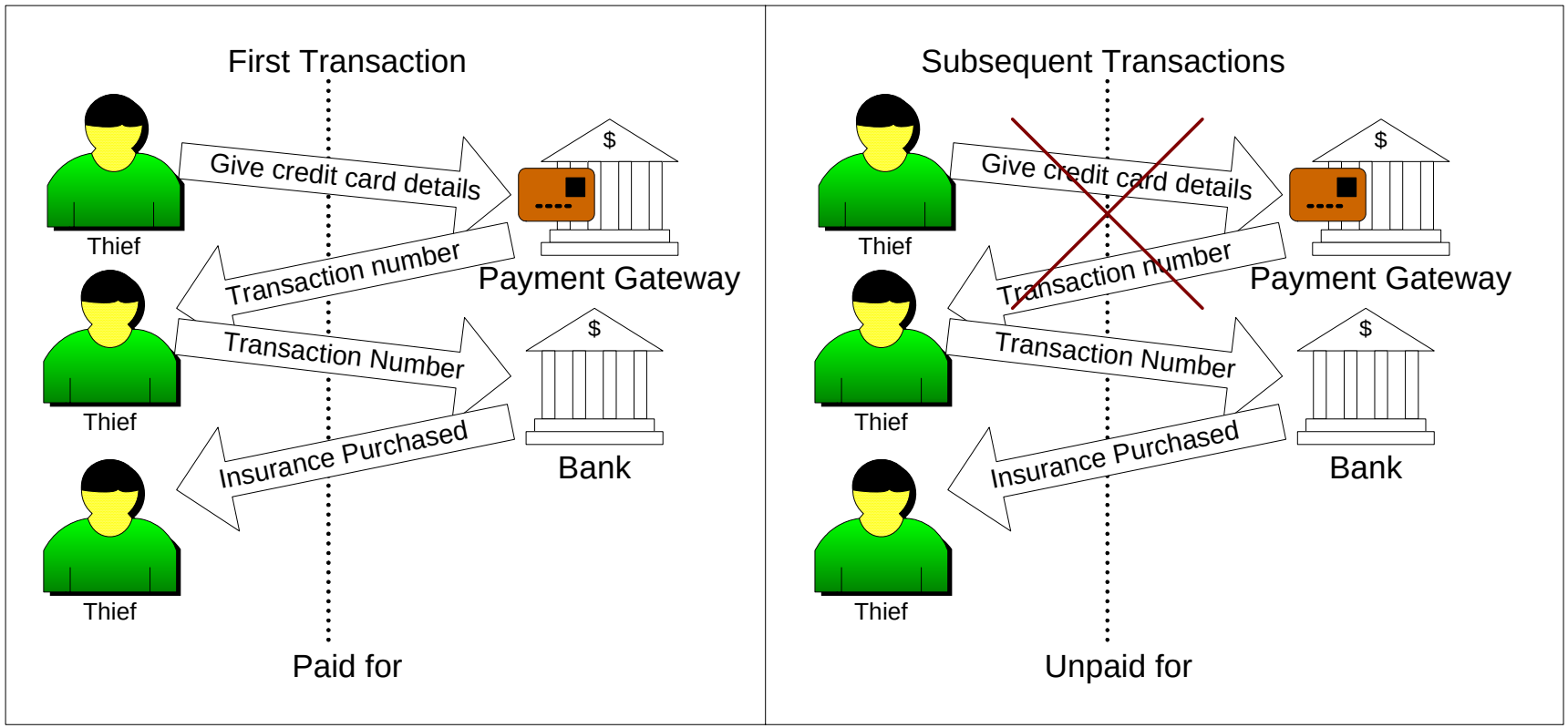


Funds from: Thief
Fees from: Thief



Funds from: Thief
Fees from: **Victim**

Purchasing Insurance for Free



This replay attack vulnerability is luckily quite rarely found

Changing Victim's Payee Information

- Log on to the internet banking using your credentials
- Go to View/Modify Payee Information
- Modify one of your payee bank account details to become your account details
- Submit the request

➔ Intercept the request

➔ Replace the payeeID with the victim's payeeID
(Fraud more effective if the payee gets paid often and a lot)

➔ Forward the request

➔ Either the application checks and return error message

➔ or it doesn't and **it replaces the payee details with yours**

➔ Every time the victim pays the payee, you get the money…

Protection against “write” logic flaws

- Protection is very **straight forward**
- Developers often **forget** to enforce it though

→Keep parameters as much as possible on the server side.

→Pass parameters by reference instead of passing them by value.

→**Verify that the data to be modified is really owned by the requesting user!**

→**Verify that the user is authorized to perform such a request!**

Internal Frauds

5.1. Running “Back-end” commands from “Front-end”.

5.2. Bypassing roles.

5.3. Bypassing authoritative boundaries.

5.4. Masquerading as a customer.

Running Back End Commands on the Front End

Back End is the administrative interface of the bank

To carry out the attack we need:

- Initial access to the back end to learn how it works

Likely

- Ex-employee
- Otherwise, brute-force or “educated guesses” sometimes work.

- The back-end commands need to be runnable on the front end

Unlikely

- Either a configuration mistake
- Or a design mistake

The attack basically uses the replay mechanism

➔ Log on to the front-end using your usual credentials

➔ Execute admin commands that you recorded previously

Bypassing Roles on the Back End

Back End has roles like admin, clerk, sales, etc…

To carry out the attack we need:

- Initial admin (access or any other role we want to attack)
  - Ex-employee, or employee that changed duties
  - Otherwise, brute-force or “educated guesses” sometimes work.

Likely

The attack basically uses the replay mechanism

- ➔ Log on to the back-end using your usual credentials
- ➔ Execute admin (or other role’s) commands that you recorded previously

Bypassing Authoritative Boundaries

Back End system allows admins to change a lot of settings…

… but some of them **supposedly** cannot be changed

- All the parameters of the application are sorted by functionality.
  - Each one has its own screen
  - Some of these settings can be changed, others cannot
- To carry out the attack, make a dummy change, and submit
  - ➔ Intercept the request
  - ➔ Make the change on one of these “unchangeable settings”
  - ➔ Forward the request
  - ➔ Either the application checks and return error message
  - ➔ or it doesn’t and **it modifies the setting**

Masquerading as a Customer

- Back End sometimes allow the bank staff to masquerade as a customer.
- All the actions are logged.
- Banking transactions involving movement of money are not available
- Some of them use a two stages authentication

Unlikely

  - Log on
  - Sign in (as the customer)

- A flaw in the authentication model allows the user to masquerade as a customer
  - Log on as “techsupport”
  - Sign in as “customer1”
  - Call directly the logout function (vs. clicking on the link)

➔ You just became “customer1” and audit trails will start failing

3rd Party Internet Banking Application Procurement

6.1. Politics involved

6.2. What you should know before buying

6.3. Role of the internal security staff

6.4. Very explicit security specifications

6.5. Role of the User Acceptance Test (UAT)

6.6. Fair contract

Politics Involved

- Bank management wants the application ready for yesterday
- Bank management feel they paid enough already
  - Expensive hardware
  - Expensive software

  → No budget left for security!
- Security team is caught in between
  - They risk their job is problems are found too late
  - They risk their job if too many problems are found and application launch late
- Security team is hated by everybody

  → **“Nobody like it when you find problem in what they’ve done”**
  - By the management
  - By the vendor
  - By the bank’s IT department

What You Should Know Before Buying

- The application **will** have security problems.
  - Internet Banking application are extremely complex and large
  - Web developers lack of security training
- The development company will say it's normal.
  - SQL injection: "– it's normal".
  - Stealing money: the bank host was supposed to check.
- They will make you pay for security fixes.
  - Since all these security bugs are "normal", fixing them is an "enhancement"
  - You have to pay for enhancements.
- They won't fix the application properly
  - Too costly to fix all the problems
  - Big time-to-market pressure.

YOU HAVE BEEN WARNED!

Role of Internal Security Staff

- Send your security staff for regular training.
- Implicate them in the buying decision process.
- Implicate them in the Application Specifications design phase.
- Implicate them in the UAT/QA test.

Very Explicit Security Specifications

Security flaws not in the list are charged to your organization

- List in details the security flaws that the vendor is forced to fix without any additional fee:
  - SQL injections
  - Buffer overflows
  - Cross site scripting
  - Privilege escalation
  - Unauthorized money transfers
  - Unauthorized charges
  - Unauthorized access to data
  - ...

SHOW THEM WHO IS THE CUSTOMER!

WARNING: THIS LIST IS NOT EXHAUSTIVE!

Role of User Acceptance Test (UAT)

- An extensive Quality Assurance plan will
  - reduce the number of regular bugs
  - reduce as well the number of security bugs
- QA test plan should include extensive input validation checks
  - will reduce by about 60% security flaws found.

- **Perform a thorough Application Security Assessment by qualified third parties during the UAT.**
  - Gives you a chance to notice all the security bugs before the application is signed-off.

Fair Contract

- Internet Banking Applications are extremely expensive
  - Annual license.
- Negotiate to obtain the source code
  - directly (sign a special license and an NDA)
  - otherwise under escrow (in case the vendor closes down).
- Negotiate to make sure the vendor is liable for fixing bugs
  - regular bugs will cost the bank money if unfixed.
  - security bugs will cost the bank money if unfixed.
- Negotiate so that the vendor pays for any additional security re-check
  - will ensure they treat security seriously.
  - will motivate them to do it right from the start.

DON'T BE A SUCKER!

Conclusion – Some Statistics

[Source: 17 last Internet Banking Application Security Assessments we conducted for major banks in the region]

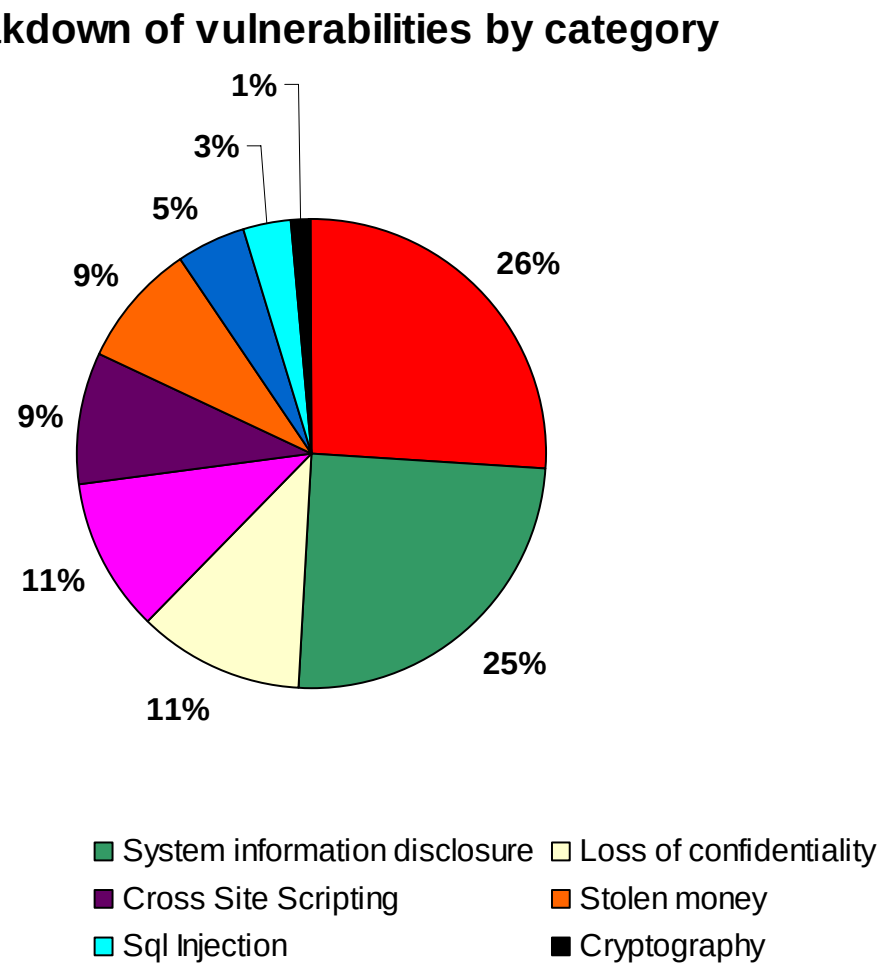
- Percentage of Internet Banking applications..
  - ...from which we stole money somehow:

100%

  - ...from which we stole personal or financial information:

100%

Statistics – continued



Statistics – continued

In these **17** application assessments

Total number of vulnerabilities found:

275

- Total number of beta-quality scripts found:

429

- Total number of unnecessary files found:

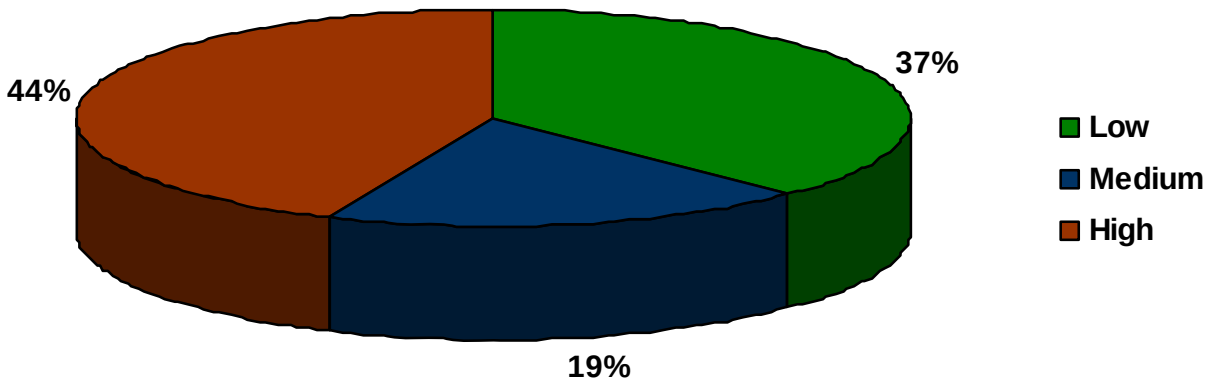
341

- Average number of vulnerabilities per application:

16

Statistics – continued

Average risk rating of vulnerabilities found



Questions and Answers

fabrice.marie@fma-rms.com
http://www.fma-rms.com/

Thank You!

FMA-RMS

fabrice.marie@fma-rms.com
http://www.fma-rms.com/