# Network Security

Course notes

Version 2013.1

# Contents

# Chapter 1

# Email Security

## 1.1 Motivation and requirements

It is easy to spoof and intercept email. Regular mail transmission over the Internet is in many ways vulnerable, e.g.

- Eavesdropping.
- Blocking.
- Replay.
- Content modification (by outsider, recipient).
- Message spoofing.
- Origin modification.
- Destination modification.
- Denial of message transmission/reception.
- Everybody can set up an email account (e.g. `GrandDukeJean@yahoo.com`).

Background.

- Email evolved on the ARPANET (1970s).
- Based on a *store-and-forward* protocol (i.e. email is not directly sent to the addressee, but passes through several intermediate nodes that temporarily store the email, possibly verify its consistency and forward the email to the next hop).
- Security was not a design requirement.
- Using SMTP (Simple Mail Transfer Protocol, 1982).
- Original e-mail features:
  - Address included in the message's body.
    Email did not rely exclusively on lower level protocols to say where the message must go.
  - Text only.
    Email messages must contain readable text, not raw binary data.

  - Accept from anywhere.
    Any delivery service or individual is allowed to submit messages that go into users' email boxes.
  - No acknowledgment.

Requirements for secure Email

- Message Confidentiality: Only intended receiver can read email.
- Sender Authenticity: The mail originates from the claimed sender.
- Message Integrity: Email cannot be altered. What the receiver sees is what was sent.
- Non-repudiation: Sender cannot deny having sent the email, receiver cannot deny having received the email.
- Protect receiver from malware embedded in email.
- Protect against the sending of unsolicited bulk messages (spam).

Security controls

- Confidentiality: Encryption
- Authenticity: Digital Signatures, (Public key crypto + Hash functions)
- Integrity: Digital Signatures, (Public key crypto + Hash functions)
- Non-repudiation: Digital signatures, "evidence of origin" (EOO), "evidence of receipt" (EOR).
- Malware: Virus scanners, email scanners.
- Spam: spam filters.

## 1.2 Solutions

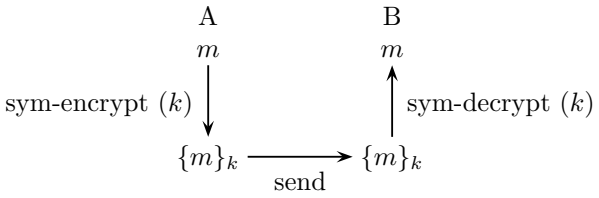
We want to send an authenticated, encrypted email. How do we achieve this? Do we encrypt first, then sign, or vice versa? What about compression?

Possible schemes:

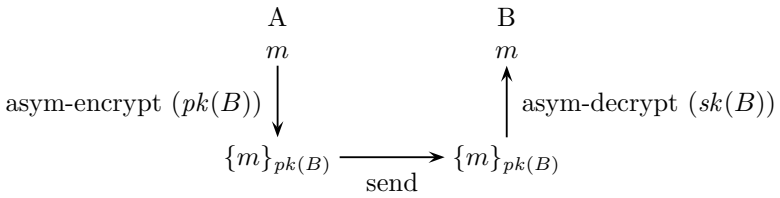
- **Asymmetric/public key:** Diffie-Hellman (ElGamal), ECC (Elliptic Curve Cryptosystem, ElGamal), RSA
- **Symmetric/secret key:** AES, IDEA, 3-DES, DES, Blowfish
- **Compressions:** zip, gzip, bzip, lzh.
- **Cryptographic Hash Function:** SHA-1, SHA-256, MD5.

### 1.2.1 Email confidentiality

**First solution for email encryption.** Encrypt message with symmetric encryption algorithm, send to friend. Problem: Both need to exchange keys in advance.




**Second solution for email encryption.** Encrypt message with asymmetric encryption algorithm, send to friend. Problem: Computationally very expensive.




**Actual solution for email encryption.**

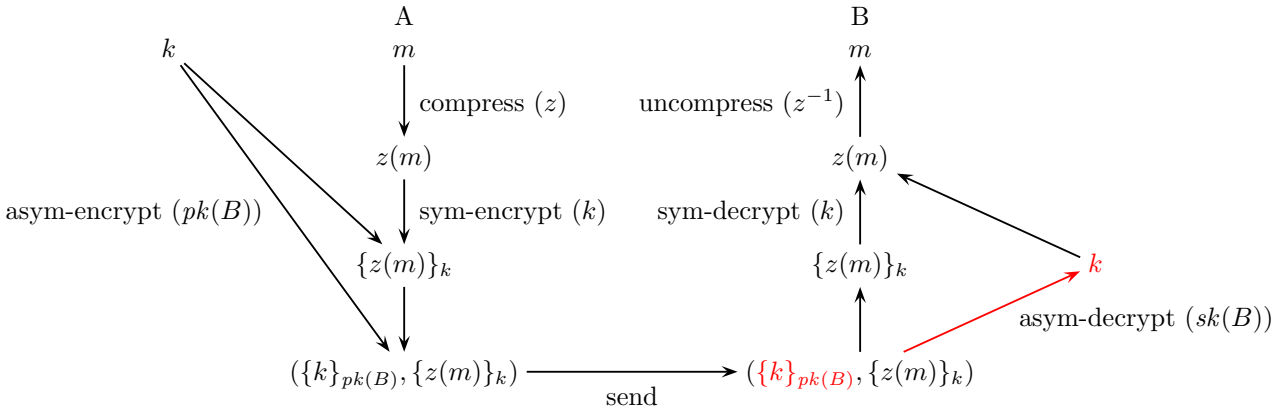
- Encryption:

  1. Generate message $m$.
  2. Generate random secret key $k$.
  3. Compress message using a compression algorithm $z$.
  4. Symmetrically encrypt message $m$ with key $k$.
  5. Asymmetrically encrypt key $k$ with recipient's public key $pk(B)$.
  6. Send send encrypted key concatenated with encrypted message.

  Given email message $m$, random symmetric key $k$ we send: $(\{k\}_{pk(B)}, \{z(m)\}_k)$

- Decryption

  1. Receive encrypted email.
  2. Asymmetrically decrypt $k$ with recipient's private key $sk(B)$.
  3. Symmetrically decrypt message with key $k$.
  4. Uncompress message with uncompression algorithm $z^{-1}$




**Compression.** When does it make sense to compress a message? Consider that compression removes redundancy from a message, thus raises the entropy of a message. Encryption raises the entropy of a message, too.

### 1.2.2 Authentication & Integrity

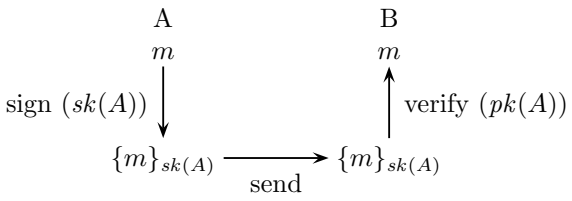
Does encryption already provide authentication and integrity protection? – NO!

Reasons:

1. With public key encryption, everyone can encrypt a message, thus sender not authenticated.
2. In some encryption algorithms, changes can be made without invalidating the encryption.
3. Messages can be replayed.

Solution: sign the messages.

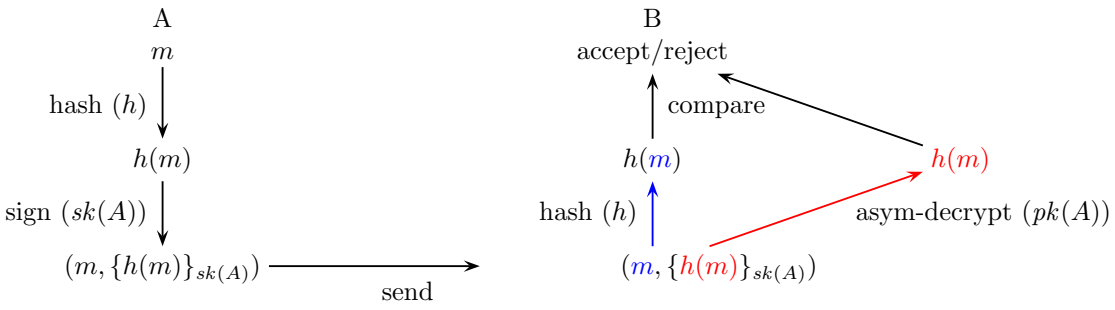
**Naive Method for Signatures.** Encrypt message using private key. Problems: Highly inefficient, not integrity protected.




**Actual Method for Digital Signatures.**

- Signature:
  1. Generate message $m$.
  2. Compute cryptographic hash code of message. (Typically 128 (MD5), 160 (SHA-1), or 256 (SHA-256) bits.)
  3. Asymmetrically encrypt the hashvalue with sender's private key $sk(A)$.
  4. Send signed hash concatenated with message.

- Verification
  1. Receive signed hashvalue and message.
  2. Decrypt signed hashvalue with sender's public key $pk(A)$.
  3. Compare this with calculated hash of received message.




## 1.3 The bootstrapping problem for trust

Remark that

- we need authentication because we don't trust the channel,
- we need to exchange keys (via a channel) to achieve authentication.

So should we send our keys over the untrusted channel?
No, that's obviously a bad idea (spoofing, modifications, etc.).

How to exchange keys in a **trusted** way?
This is called the *bootstrapping problem*: we can't trust the channel without keys, and we cannot trust the channel to deliver the keys correctly. The solution: use another channel!

All people who wish to communicate meet in real life. They

1. prove to each other that they really are who they claim to be (e.g. show passport), and then
2. ask the other to sign their public key.

Now if later they send a signed email, the other can verify the signature on the email, and can verify that he/she decided to *trust* the key.

Note that a similar argument may be made for confidentiality: how do you know who owns the decryption key to a particular public key $PK$? Again, no way to start this trust (i.e., to "bootstrap") via the untrusted channel – so use a different channel.

## 1.4 Implementations

- PGP = Pretty Good Privacy (Phil Zimmermann, 1991, originally designed as a human rights tool)
- S/MIME = Secure / Multipurpose Internet Mail Extensions, 1995)

### 1.4.1 PGP

(Similar to GPG = GNU Privacy Guard,1997)
Zimmermann's goals:

- Select best available crypto algorithms.
- Integrate in general purpose, os/processor-independent tool.
- Tool and sources freely available.

Exporting strong crypto from the USA was illegal, but Zimmermann found a back door: exporting books was allowed. Therefore he published the source code in a book and developed excellent OCR (Optical Character Recognition) software, which he also put into the book. The OCR software could be used to read the book and translate it automatically into (verified) code.

What we have described is essentially how GPG / PGP work. In particular, GPG / PGP use the following cryptographic algorithms:

**Public key cryptosystems** ElGamal, DSA, RSA

**Symmetric key cryptosystems** AES, 3DES, Blowfish, Twofish, CAST5

**Hash functions** MD5, SHA-1, RIPE-MD-160, MD2, TIGER

**Compression functions** ZIP, ZLIB

**Email Compatibility** Radix 64 conversion (MIME base64)

Key management for PGP. A user can have several $(pk, sk)$ key pairs. How to determine which key is used by the sender?

- Every key pair has a (hopefully) unique KeyID, consisting of the least significant 64 bits of the public key. This keyID is sent along with the message.
- Every user has one (or more) UserID's. This is typically the user's e-mail address.

Each user maintains:

- Private key-ring to store his own key pairs $(KeyID, pk, \{sk\}_{h(p)}, UserID)$, where $h(p)$ is the hash of his passphrase.
- Public key-ring to store other users' public keys $(KeyID, pk, UserID)$.

Where to obtain the other users' public keys?
Website, e-mail, key server, ...
Problem: fake/compromised keys give an attacker the opportunity to

- fake signatures
- read encrypted messages

Therefore, one normally verifies the public key (or rather its *fingerprint*) via a trusted channel. Trust in someone else's key is not a yes/no question. For instance, you obtain A's key from B. You trust B, but you know that he is a bit sloppy sometimes.

Therefore, we add a *key legitimacy field* to every key in the public key ring. This field contains the level of trust. This level can be *unknown*, *untrusted*, *marginally trusted*, *completely trusted*.

Further, we add a signature of the public key. This expresses that the signer trusts this public key. We also add a *signature trust field* to express our trust in the signer.

This allows to specify your own trust policies, e.g. *You completely trust someone's public key if it is signed by someone you completely trust or by three people you trust marginally.*

Possible further topics: key signing parties and key revocation.

# Chapter 2

# Security Protocols

## 2.1 Motivation

*"Security protocols are three-line programs that people still manage to get wrong."*

— Roger Needham.

A protocol is a set of interaction rules designed to guarantee functionality. A *security* protocol is a set of interaction rules designed to guarantee *specific security properties* as well as intended functionality.

### 2.1.1 A hierarchy of building blocks

| Levels | Examples | Sample attacks |
|---|---|---|
| **Implementation** | Web browser, OSI protocol stack | Buffer overflows, weak random numbers |
| **Systems** | Voting, Road pricing, Mobile Internet | Social engineering, burning ballot |
| **Security Protocols** | IPV4, Password-based authentication | Man-in-the-middle, replay attack |
| **Cryptography** | ElGamal, AES | Collision attack on hash functions |
| **Mathematics** | Number theory, Fields | Non-strong primes |

### 2.1.2 Security requirements

A number of security requirements that protocols may need to satisfy:

- Confidentiality (keeping messages secret).
- Authentication (proving identity).
- Integrity (keeping messages uncorrupted).
- Privacy (not learning information about someone).
- Unlinkability (not being able to link two events together).
- Non-repudiation (not being able to deny having taken an action).

In this chapter we take a close look at confidentiality, authentication, unlinkability and non-repudiation.

### 2.1.3 Assumptions

We study security protocols in isolation from the other levels (crypto, system, implementation). This gives the advantage of *separation of concerns*. Thereto, we assume that the crypto-level provides cryptographic primitives, like hashing and signing, which are perfect, the *perfect cryptography assumption*. The implementation of the cryptographic primitives then can be considered as a black box, and we can use idealized abstractions of these primitives, the *black-box approach*. We will denote the hashing of a message $m$ with hash function $h$ simply by $h(m)$ and assume that it satisfies the required properties, like collision resistance. Symmetric encryption of a message $m$ with key $k$ will be denoted by $\{m\}_k$. Asymmetric encryption with $a$'s public key will be denoted by $\{m\}_{pk(a)}$.

### 2.1.4 What goes wrong

Security protocols are hard to get right. Sources of errors include:

- Insecure protocols.
- Incorrect cryptographic primitives.
- Bad implementations.
- Changed assumptions on environment.

### 2.1.5 Security misconceptions

Some misconceptions with respect to security:

- "Our application is secure because we are using the most up-to-date cryptographic algorithms."
- "We keep our communication protocol secret, so nobody will ever be able to attack it and we don't have to worry about security of the protocol."
- "My engineers looked at the protocol for a while and could not find errors, so we can safely use it."
- "Now that we have finished the design and specification of our system, we can start adding security features."
- "In order to be sure, we apply double encryption."
- "We have found this security protocol in literature, so we can safely embed it in our system."

Cryptography is like a lock. No matter how good a lock is, if it is used incorrectly, it doesn't protect (see Figure 2.1.5).

Figure 2.1: Cryptography is like a lock (photo by C.J.F. Cremers)

# Chapter 3

# Web Security

## 3.1 Web Security

### 3.1.1 Introduction

[1] Today's web applications are often vulnerable to attacks. There are several reasons why many web applications contain vulnerabilities. Programmers are generally not aware of good security practices, security is hard to get right, security is inconvenient. Moreover, there is no direct return on investment on secure applications. There is also the predominant thinking that one web applications vulnerability does not have the capability of inflicting major damage.

Table 3.1 gives a non-exhaustive overview of today's web vulnerabilities. Web vulnerabilities can be classified by the victim (either a remote system, or a local system) and the target (either data, or a complete system).

It may not come as a surprise that web vulnerabilities cannot be prevented completely. However, one must be aware of the mistakes and vulnerabilities that are present, and adapt his web applications to minimize the chance that they are compromised.

The subsequent sections present common web vulnerabilities in today's web applications. The common divisor of these vulnerabilities is that they rely on input from the attacker that is executed. Therefore, a web application should always sanitize input provided by the user.

To understand why input validation is necessary, it is essential to understand the communication between a browser and a remote server. The communication starts by the browser issuing an HTTP GET command to the server:

```
GET /members/sasa/fun/ HTTP/1.1
Host: satoss.uni.lu
```

The server then responds with:

Table 3.1: Classification of web vulnerabilities

| | Data | Computer System |
|---|---|---|
| Remote System | SQL injection | Apache Buffer Overf. |
| Local Computer | Resp.-splitting XSS, Phishing cookie pois. CSRF | Email attachments, WMF/TIFF Vulner. Trojans, Malware |

```
HTTP/1.1 200 OK
Date: Thu, 24 Nov 2011 10:39:33 GMT
Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny13 with Suhosin-Patch
mod_ssl/2.2.9 OpenSSL/0.9.8g
Last-Modified: Wed, 04 Feb 2009 14:25:25 GMT
ETag: "57840-7f-462188f043f40"
Accept-Ranges: bytes
Content-Length: 127
Content-Type: text/html

<HTML>
<HEAD><TITLE>Demo Website</TITLE></HEAD>
<BODY onload='alert("Printer on Fire!");'>
<H1>Hello World</H1>
</BODY></HTML>
```

The response is interpreted by the browser in three steps:

1. The browser reads the header:

   ```
   HTTP/1.1 200 OK
   Date: Thu, 24 Nov 2011 10:39:33 GMT
   Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny13 with Suhosin-Patch
   mod_ssl/2.2.9 OpenSSL/0.9.8g
   Last-Modified: Wed, 04 Feb 2009 14:25:25 GMT
   ETag: "57840-7f-462188f043f40"
   Accept-Ranges: bytes
   Content-Length: 127
   Content-Type: text/html
   ```

2. The browser reads and displays the HTML code:

   ```
   <HTML>
   <HEAD><TITLE>Demo Website</TITLE></HEAD>
   <BODY onload='alert("Printer on Fire!");'>
   <H1>Hello World</H1>
   </BODY></HTML>
   ```

3. The browser interprets and executes scripts in the content:

   ```
   alert("Printer on Fire!");
   ```

### 3.1.2 Cross-site scripting (XSS)

Cross site scripting allows attackers to execute scripts in the user's browser. This may result in hijacked user sessions, defaced web sites, hostile content in web sites, phishing attacks, and hostile browser take-overs. The malicious script is usually JavaScript, but may be any scripting language that is interpreted by the victim's browser. Cross site scripting flaws occur whenever an application takes data that originates from the user and does not validate the user input.

The attacks are usually implemented in JavaScript, which is a powerful scripting language. Using JavaScript allows attackers to manipulate any aspect of a rendered page, such as adding a login box which forwards credentials to a hostile site. Another possibility is to perform a phishing attack on the user (see Section 3.1.6). The evolution of JavaScript malware, finding its way into more and more attackers toolboxes, has made finding and fixing this vulnerability more vital than ever.

XSS attacks can be protected against by validation of all incoming data ("whitelist validation") and appropriate encoding of all output data. Validation allows the detection of attacks, and encoding prevents any successful script injection from running in the browser.

### 3.1.3 Injection flaws

Injection flaws are common in web applications. There are many types of injections, for instance SQL, LDAP, XPath, XSLT, XML, and OS Command injections. Injection flaws occur when user-supplied data is passed to an interpreter as part of a command or query. The most common type of injection flaw allows for SQL injection. Structured Query Language (SQL), is a computer language designed for the retrieval and management of data in relational database management systems, database schema creation and modification, and database object access control management.

A typical SQL query looks like

```
SELECT field FROM table WHERE condition;
```

For instance, if we have a table of all students and their grades, we might want to issue the following command

```
SELECT student FROM allstudents WHERE grade < 10;
```

to list all those students whose grade is below 10.

A classic way to verify a username and password provided by a user is to search the database for that username/password. If the database returns a non-empty set, the user has provided a valid pair, and passes authentication. A common way to do this is to access the database from a PHP script using the following code:

```
$sql    = "SELECT * FROM table WHERE username = '" . $user . "'
          AND password='" . $password . "';";
$result = mysql_query($sql);
```

A simple SQL injection that would allow access to the website could be to provide `admin` as username and `anypassword' OR '1'='1` as password. Giving such values would result in the following SQL query being executed:

```
$sql    = "SELECT *
           FROM table
           WHERE username = 'admin'
           AND password='anypassword' OR '1'='1';
```

The latter part of the injection (`' OR '1'='1`) ensures that the WHERE-clause is always satisfied. This would allow the attacker to pass the authentication mechanism and login to the system without knowing a valid username/password combination.

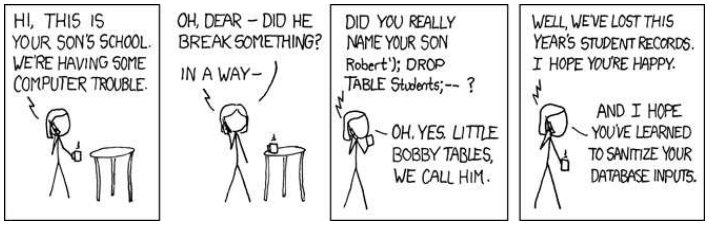
Although passing the authentication scheme is undesired, much more damage can be done by exploiting the fact SQL queries can be composed using a semicolon `;`. For instance, the following input for *password* to the above query would add a user "Ton" to the database with password "12345".

```
anypassword' OR '1'='1'; INSERT INTO table VALUES ('Ton','12345');--
```

Since the values of `$user` and `$password` are entirely under the control of the attacker, any malicious input can be passed on to the database.

Any application that allows users to input data which is subsequently passed on to another system is potentially vulnerable to injection flaws. The main solution to this problem is to sanitize the data (for instance by disallowing certain characters in user input), before passing it on to another system.

SQL Injection has been at the center of some of the largest credit card and identity theft incidents. Today's backend website databases store highly sensitive information, making them a natural, attractive target for malicious hackers. Names, addresses, phone numbers, passwords, birth dates, intellectual property, trade secrets, encryption keys and often much more could be vulnerable to theft. With a few well-placed quotes, semi-colons and commands entire databases could fall into the wrong hands.




### 3.1.4 Cross-site request forgery (CSRF)

Cross-site request forgery is not a new attack, but it is gaining in popularity. A CSRF-attack forces a logged-on browser to send a request to any web site of the attacker's choosing, which then performs the chosen action on behalf of the victim. The attack works by including a link into a web page that accesses another web site to which the user has logged in. The attack allows attackers to make an HTTP request to e.g. the victim's bank, blog, or web mail. The following tag in any web page viewed by the victim will generate a request to the web page that logs out the user:

```
<img src="http://www.example.com/logout.php">
```

If an online bank allowed its application to process requests without explicitly verifying the user's credentials, the following code asks for a transfer of funds:

```
<img src="http://www.example.com/transfer.php?toIBAN=123
 &toBIC=456&amount=500">
```

Although XSS flaws are not required for CSRF attacks to work, any web site that is vulnerable to XSS attacks is also vulnerable to CSRF attacks. When building defense against CSRF attacks, eliminating XSS vulnerabilities in a web site is essential, since such flaws can be used to get around CSRF defense mechanisms.

The only protection against CSRF attacks is that applications do not rely on credentials that are automatically submitted without the user's knowledge.

### 3.1.5 Information leakage

Applications can unintentionally leak information about their configuration, internal working, or violate privacy through a variety of application problems. Examples of such information are developer comments, user information, internal IP addresses, source code, software version numbers. While information leakage itself does not have to be a problem, the information may be used by an attacker to launch, or even automate more powerful attacks. Examples of information leakage:

- Detailed error handling such as failed SQL statements, or other debugging information.
- Functions that display different results based on different inputs. For example a failed login should display the same message (e.g. `"login failed"`) irrelevant of whether username, password, or both were incorrect.

### 3.1.6 Phishing attacks

Phishing is an attempt to criminally and fraudulently acquire sensitive information by masquerading as a trustworthy entity in an electronic communication. The first phishing technique was described in detail in 1987. Phishing was first applied on a large scale in the early 90s. While early phishing attempts were targeted at large groups of users, recent phishing attempts are becoming more and more targeted at individuals using additional information of the user that is attacked.

In a phishing attempt, the attacker tries to spoof a legitimate web site, with the aim of obtaining data the victim would normally supply to the original web site. This data could be usernames and passwords or credit card information.

The first part of a phishing attack is to fool the user into visiting a malicious web site. Common techniques are to make the anchor text for a link appear to be valid, while the link actually goes to the malicious web site. Another common technique is to use misspelled URL's, the use of subdomains (e.g. `http://www.bcee.lu.com` or `http://www.bcee.com.lu` when the attacker possesses resp. the domains `lu.com` or `com.lu`), or the use of IP addresses instead of URL's. Another, more dangerous attack involves poisoning the DNS cache of the user. If the attacker succeeds in poisoning the DNS cache of the user, the user will be redirected to the wrong server.

After fooling the user into visiting the malicious web site, the attacker might try to alter the address bar. He can either use JavaScript to alter the address bar, place a picture of a legitimate URL over the address bar or close the original address bar and open a new one with the URL of the legitimate URL.

In general phishing attacks are performed through e-mail or instant messaging communication.

### 3.1.7 Browsers compromising privacy

Recently, two examples have shown that websites with CSS can be used to violate the privacy of the user visiting the website. The website uses the fact that the browser stores a history of visited pages. Depending on whether a user has visited a page that is linked to from a browser, the color of the hyper link is changed. The following two web sites show whether a user visited a page or not:

`http://ha.ckers.org/weird/CSS-history-hack.html`
`http://ha.ckers.org/weird/CSS-history.cgi`

The second web site shows how a hostile web site may abuse this information: it includes an image iff the user has visited the page. In doing this, the browser essentially tells the web site which pages have been visited.

### 3.1.8 Predictable Resource Location (PRL)

Over time, many pages on a website become unlinked, orphaned, and forgotten - especially on websites experiencing a high rate of content and/or code updates. These Web pages sometimes contain payment logs, software backups, post dated press releases, debug messages, source code - nothing, or everything. Normally, the only mechanism protecting the sensitive information within is the predictability of the URL. Automated scanners have become adept at uncovering these files by generating thousands of guesses. However, although a scanner can guess at a filename, it has no contextual reasoning to tell if the data received is sensitive or how valuable it might be. Humans need to make this determination.

### 3.1.9 Sources

These notes are among others based on the following:

- OWASP Top 10 Project[1].

[1] `http://www.owasp.org/index.php/OWASP_Top_Ten_Project`

- MITRE vulnerability trends[2].

[2] http://cwe.mitre.org/documents/vuln-trends/index.html

# Bibliography

[1] Adi Shamir. On the power of commutativity in cryptography. In *ICALP*, pages 582–595, 1980.