Contents

1 Email Security

1.1 Background

Protocols:

Architecture and message fields:

- Internet Mail Architecture - RFC 5598
- Internet Message Format - RFC 5322

Deliver messages (client-to-server, server-to-server):

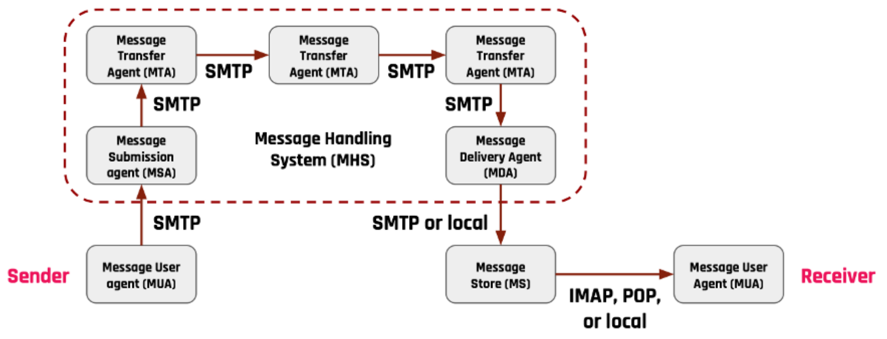
- Simple Mail Transfer Protocol (SMTP) - RFC 780 RFC5321
- Extended Simple Mail Transfer Protocol (ESMTP) - RFC 1869
- Message Submission - RFC 2476
- SMTP Service Extension for Authentication (SMTP-AUTH) - RFC 2254

Retrieve messages (server-to-client):

- Internet Message Access Protocol (IMAP) - RFC 3501
- Post-Office-Protocol (POP) - RFC 1939

1.1.1 Internet Mail Architecture

The e-mail architecture is constituted by several components:


Message Transfer Agent (MTA)
SMTP
Message Transfer Agent (MTA)
SMTP
Message Transfer Agent (MTA)
SMTP
SMTP
Message Submission agent (MSA)
Message Handling System (MHS)
Message Delivery Agent (MDA)
SMTP
SMTP or local
Sender
Message User agent (MUA)
Message Store (MS)
IMAP, POP, or local
Message User Agent (MUA)
Receiver

- Message User Agent (MUA): it's actually the client used by end user.
- Mail submission agent (MSA): it sends the message to the recipient using a MTA. It interacts with the MUA to receive commands, store locally the information that are then passed to the MTA to create the real email.

- Message transfer agent (MTA): it's what we typically call the mail or SMTP server. In general, we can have several MTA in order to reach the MTA of the email recipient. When the email arrives to such MTA, it recognizes that it doesn't need to relay the message anymore and deliver the message to a MDA.
- Mail delivery agent (MDA): it can possibly store the message on a MS. Typically, the MDA and MS are implemented by a single process.
- Message store (MS): it's a location where the MDA can store the message received from the sender.
- Message user agent (MUA): once the message is stored, it can be read or retrieved by the user using its client.

Implementations have often merged different roles into the same software component:

- MSA and MTA
- MTA and MDA

Nowadays, we don't have a MUA but we tend to use a web client (e.g., GMAIL), which often uses proprietary protocol to communicate with the MSA/MTA/MDA.

How to know the next hop?

The sender's server performs a DNS query to reach the server of the destination. A DNS 'mail exchange' (MX) record directs email to a mail server. The MX record indicates how email messages should be routed in accordance with the Simple Mail Transfer Protocol (SMTP, the standard protocol for all email). SMTP uses a port to forward emails, but the MX record does not report the port. By default, port 25 is used... Consumer ISP usually blocks this port... hence you cannot host your mail server.

1.1.2 Retrieving emails -protocols

With POP protocol, client contact the MS and say "give me a copy of all emails received from this date to this date". Next, it retrieves such emails and then they are deleted from the MS (there is also an option to leave the email there). The concept of IMAP is quite different; in fact the MS is designed to store emails. When we access the MS to read emails, we ask it to make a copy of a specific message we want to read. If we want to move emails or performs any other operation, we will sends commands to the MS, but the operation is performed on the MS. What we see with IMAP on our client, is somewhat a view of our message box on the MS.

POP:

- supports download and delete operations
- messages are locally stored using, e.g., a mbox format

- **messages are fetched by the client and then deleted from the server**
- connection established only when fetching messages
- POP3 supports SSL/TLS (default port: 995)

IMAP:

- the idea is to permit even different clients to access the same mail box
- **a copy of the message is left on the server even when fetched by a client**
- the connection is kept on while the client is browsing the email (faster response time)
- message state information: read, replied, or deleted
- server-side searches (which could increase the load...)
- IMAP supports SSL/TLS (default port: 993)

1.1.3 Delivering email -protocols

SMTP exchanges messages using the message envelope separate from the message itself. The protocol also defines the way sender and receiver can be identified, typically using the standard email addresses. The client need to use the protocol to synchronize its local view with respect to the reality that is given by the status of the MS. MTAs speaks between themselves using the SMTP protocol. Next, this protocol was extended to include some enhanced features such as encryption, which goes under the name of ESMTP. This extension is designed such that all software compatible with ESMTP will be also compatible with SMTP.

SMTP:
- main commands: HELO, MAIL FROM, RCPT TO, DATA, VRFY/EXPN, TURN, AUTH, RSET, HELP, QUIT

ESMTP:
- new commands: EHLO, STARTTLS, SIZE, 8BITMIME, ATRN, CHUNKING, DSN, ETRN, PIPELINING, SMTPUTF8, UTF8SMTP
- support for encrypted connections on SSL/TLS

1.1.4 Message format

Typically an email is made up of three components:

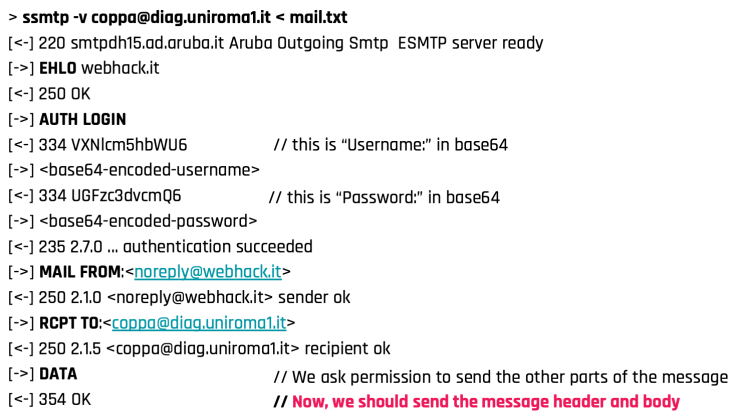
- message envelope: it's used by MTAs to correctly relate the message from the sender MTA to the recipient MTA. It is designed to contain minimal info about sender/receiver that can be quickly processed by SMTP servers along the route (possibly multi hops).

- message header: where we write who is sending the email, who is the recipient, i.e. they are meta information. In particular, it's structured into fields such as From, To, CC, Subject, Date and other information about the email. It's separated from the message body by a blank line. These info are mostly useful to the last SMTP server.

- message body: it's the text of the message that we want to send, typically unstructured.

Main fields of envelope:
- MAIL FROM: bounce address, i.e., where to send back the message in case of failure. Alternative names: RETURN PATH, REVERSE PATH, BOUNCE ADDRESS
- RCPT TO: receiver's address, which may be more than one in case of, e.g., CC/BCC. You may use telnet to send these commands. However, most SMTP servers nowadays requires a SSL connection hence it would not work. One solution is to use OpenSSL.

Example



```
> ssmtp -v coppa@diag.uniroma1.it < mail.txt
[<-] 220 smtpdh15.ad.aruba.it Aruba Outgoing Smtp  ESMTP server ready
[->] EHLO webhack.it
[<-] 250 OK
[->] AUTH LOGIN
[<-] 334 VXNlcm5hbWU6            // this is "Username:" in base64
[->] <base64-encoded-username>
[<-] 334 UGFzc3dvcmQ6            // this is "Password:" in base64
[->] <base64-encoded-password>
[<-] 235 2.7.0 … authentication succeeded
[->] MAIL FROM:<noreply@webhack.it>
[<-] 250 2.1.0 <noreply@webhack.it> sender ok
[->] RCPT TO:<coppa@diag.uniroma1.it>
[<-] 250 2.1.5 <coppa@diag.uniroma1.it> recipient ok
[->] DATA                        // We ask permission to send the other parts of the message
[<-] 354 OK                      // Now, we should send the message header and body
```

There are mandatory header fields such as :

- From: The email address, and optionally the name of the author(s)

- Date: The local time and date when the message was written

- To: The email address(es), and optionally name(s) of the message's recipient(s). Indicates primary recipients (multiple allowed)

- Subject: A brief summary of the topic of the message. Certain abbreviations are commonly used in the subject, including "RE:" and "FW:"

TO: vs RCPT TO: They could be different!
FROM: vs MAIL FROM: They could be different! Why? A common answer is to support automated processes sending mail (e.g., mailing lists) and to send the same message to different receivers (e.g., cc).

Other common header fields are the following :

- Bcc : Blind Carbon Copy; with this option we can add addresses to the SMTP delivery list but they will not appear in the message data, i.e. they are invisible to other recipients.

- Cc : Carbon copy; it's similar to the previous field, but in this case all the addresses in the SMTP delivery list are listed in the message data.
- Content-Type : Information about how the message is to be displayed, e.g., a MIME type.
- Content-transfer-encoding: encoding used by the content
- Message-ID: Automatically generated field; used to prevent multiple delivery and for reference in In-Reply-To:
- References: Message-ID of the message that this is a reply to, and the message-id of the message the previous reply was a reply to, etc.
- Reply-To: Address that should be used to reply to the message
- In-Reply-To: Message-ID of the message that this is a reply to. Used to link related messages together. This field only applies for reply messages
- Sender: Address of the actual sender acting on behalf of the author listed in the From: field (secretary, list manager, etc.)
- Archived-At: A direct link to the archived form of an individual email message

Together with the headers that we talked about, SMTP defines a lot of trace information that are included in the message as it's relayed toward the destination, which are also saved in the header.

- Received: when an SMTP server accepts a message it inserts this trace record at the top of the header (last to first)
- Return-Path: when the delivery SMTP server makes the final delivery of a message, it inserts this field at the top of the header
- Auto-Submitted: is used to mark automatically generated messages
- For security checks, other fields may be inserted

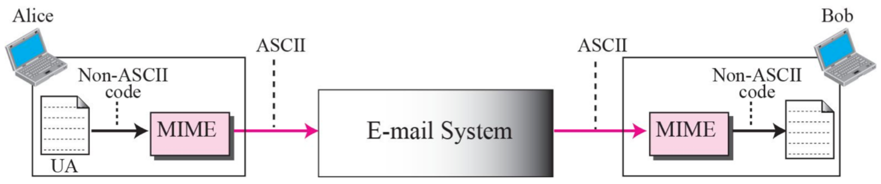
An open mail relay is SMTP server configured in such a way that it allows anyone on the Internet to send e-mail through it, not just mail destined to or originating from known users. This used to be the default configuration in many mail servers; indeed, it was the way the Internet was initially set up, but open mail relays have become unpopular because of their exploitation by spammers and worms. Many relays were closed, or were placed on blacklists by other servers.

1.1.5 MIME

By default, SMTP supports only ASCII content. Another standard often used in email system is MIME (Multipurpose Internet Mail Extensions), but it's also used in the web for other reasons. It's a standard that explain us how to encode complex content in more simpler containers. It allows to include : text in character sets other than ASCII, non-text attachments, message bodies with multiple parts and header information in non-ASCII character sets. MIME allows to encode the content with:
- Base64
- Quoted-Printable (QP)
- Binary

Or, when the ESMTP server advertises 8BITMIME, we could directly send 8-bit data. For compatibility reasons, MIME with (BASE64, QP) is often the preferred choice.

How does MIME works ?



The idea is the following one: when the user want to send a message through the email system that contains some information that are not in the ASCII character set, the client encodes this information using MIME in a transparent way. First it analyze the content that is provided by the user, checks how each part of the content should be encoded and set up a MIME compliant message with the correct encoding of each part. Then this MIME encoded message is fully expressed using only 7 bit ASCII encoding and then it can be sent via the email system. At the other end of the email system when the message is received, the client of the receiver apply the opposite process, decoding the MIME content and correctly rendering the content of the message.

The important point is that the MIME encoding of the message body is completely transparent to the email system, because MIME provides a way to encode and decode this content at the endpoint, and everything is completely transparent to the email system since it never look at the message content. The kind of encoding used for MIME is defined through MIME headers which contains the version information, the type of the content that will be provided by the email body and other meta data such as encoding type, content-id and content-description. Some of the most common MIME types are text/plain for plain text data, text/html for html content, etc.

Base64 is the standard way to encode non textual content in MIME messages. The idea is that through base64 we start from a binary content and obtain as target a string that is limited to A–Z, a–z, 0–9, +, / character set. This is done by encoding each block of 6 bits into a 8-bit

ASCII values. BASE64 is simple, yet it comes with space overhead (+25% bits).

Quoted-printable encoding: any 8-bit byte value may be encoded with 3 characters: an '=' followed by two hexadecimal digits (0–9 or A–F) representing the byte's numeric value
- non 8-bit byte values are ASCII chars from 33 to 126 (excluded 61, the '=' sign)
- special cases for SPACE and TAB
- more space efficient than BASE64 but it not space uniform

Multipart subtypes The MIME standard defines various multipart-message subtypes, which specify the nature of the message parts and their relationship to one another. The subtype is specified in the Content-Type header field of the overall message. The most common subtypes are:

- Mixed. For sending files with different "Content-Type" headers.
- Digest. To send multiple text messages.
- Message. Contains any MIME email message, including any headers
- Alternative. Each part is an "alternative" version of the same (or similar) content (e.g., text + HTML)

For example, a multipart MIME message using the digest subtype would have its Content-Type set as "multipart/digest".

1.2 Security problem

Three main risks from emails:

1. Email spamming: unsolicited electronic messages (like advertisements)
2. Email tracking: emails can be used to track user actions thus leading to privacy issues
3. Email phishing: social engineering attacks based on a fraudulent ("spoofed") message. (someone sends messages pretending to be someone else or with dangerous content in order to take some informations)

1.2.1 SPAM

Spamming is the use of messaging systems to send multiple *unsolicited messages* (spam) to large numbers of recipients. The goals of spam are:

- sell products/services (aggressive marketing)
- sell low-qualities/fake/expired goods/medicines (low prices)

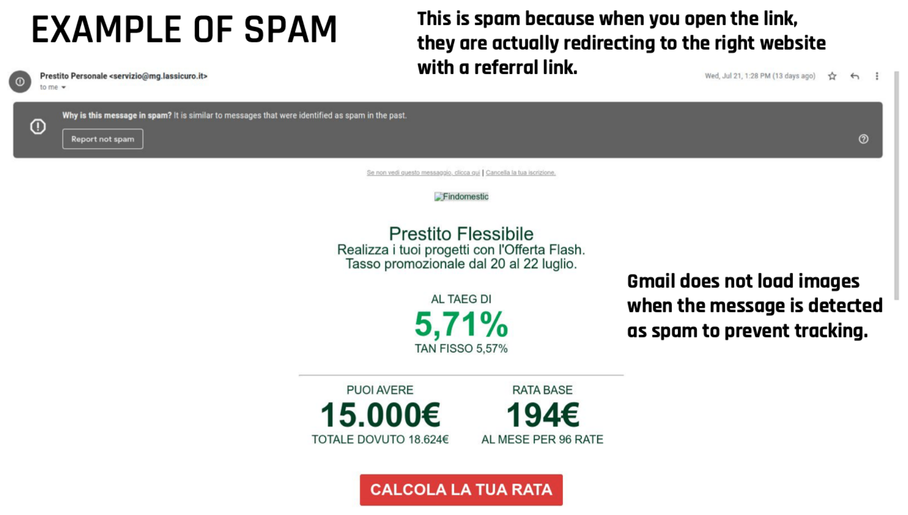
- distribute/spread malware (viruses, worms, Trojan horses, backdoors, rootkits etc.) and grayware (adware,spyware, dialers etc.)
  - computer can be enrolled/controlled for participation in (future) attacks
  - Internet activity (browsing, instant messaging and other social activity) can be monitored, users can be profiled
  - audio/video sessions can be recorded
  - collect (any) data on you and on your contacts (databases are built to the purpose of digital identity thefts)

- phishing
  - username/password stealing, credit card data capture, frauds etc.
  - often based on malicious links

- validate e-mail addresses
  - can be re-sold at a higher price
  - based on HTML images and links

A brief list of suggestions to follow when using the email system:

- Disable html message or, at least, disable download of remote images
- Don't click links, since we could redirect us to bad web sites containing malware
- Don't open unknown attachments since they may contain malware
- Activate local anti-spam filter
- Don't participate with chain letters
- Protect and respect privacy of other recipients
- Even if non-Windows user, activate antivirus for protecting your recipients
- Don't provide your personal data
- Don't click ”delete me” link.


EXAMPLE OF SPAM

This is spam because when you open the link, they are actually redirecting to the right website with a referral link.

Prestito Personale <servizio@mg.lassicuro.it>
to me

Wed, Jul 21, 1:28 PM (13 days ago)

Why is this message in spam? It is similar to messages that were identified as spam in the past.

Report not spam

Se non vedi questo messaggio, clicca qui | Cancella la tua iscrizione.

Findomestic

Prestito Flessibile
Realizza i tuoi progetti con l'Offerta Flash.
Tasso promozionale dal 20 al 22 luglio.

AL TAEG DI
5,71%
TAN FISSO 5,57%

PUOI AVERE
15.000€
TOTALE DOVUTO 18.624€

RATA BASE
194€
AL MESE PER 96 RATE

CALCOLA LA TUA RATA

Gmail does not load images when the message is detected as spam to prevent tracking.

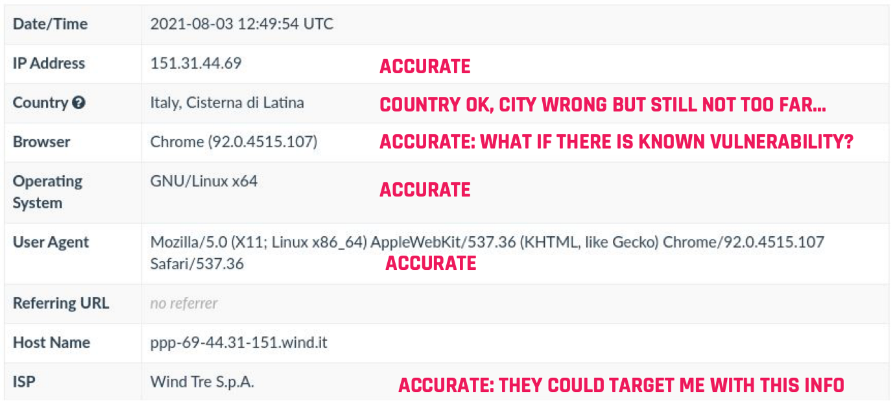
1.2.2 Email tracking

By reading an email, you may reveal sensitive information:
- whether the email was read: the email address is valid and the user likely read the content.
- the IP address of the victim, which may be used to perform direct attacks or know the approximate location (country and city)
- other info (browser, device, etc.) about the user sent by the browser when performing a request

This can be easily achieved by embedding:

- external images: a unique URL is associated to each email message and the attacker only needs to check the access log on its server.
- a shortened URL: if the user opens the URL, a page tracks its info and immediately redirect him to a valid page
- an "unsubscribe" URL: expert users may trick into this one...

Example: using a tracking service we can derive a lot of information about a URL:



Date/Time	2021-08-03 12:49:54 UTC	
IP Address	151.31.44.69	ACCURATE
Country	Italy, Cisterna di Latina	COUNTRY OK, CITY WRONG BUT STILL NOT TOO FAR...
Browser	Chrome (92.0.4515.107)	ACCURATE: WHAT IF THERE IS KNOWN VULNERABILITY?
Operating System	GNU/Linux x64	ACCURATE
User Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.107 Safari/537.36	ACCURATE
Referring URL	*no referrer*	
Host Name	ppp-69-44.31-151.wind.it	
ISP	Wind Tre S.p.A.	ACCURATE: THEY COULD TARGET ME WITH THIS INFO

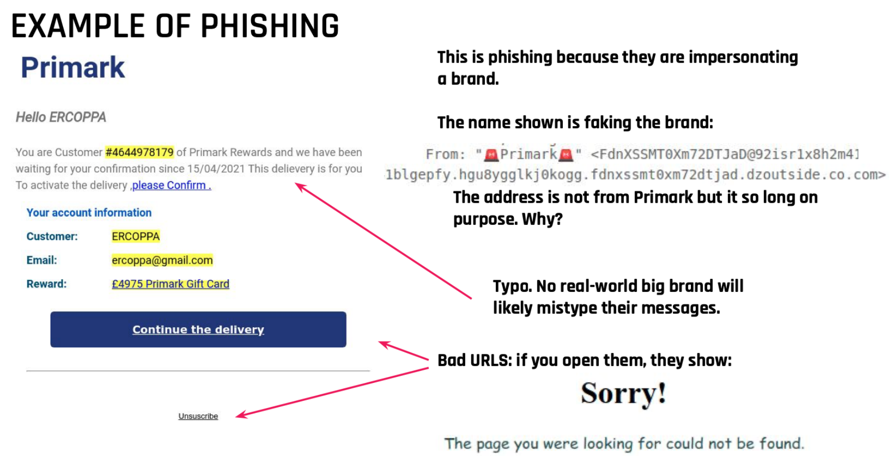
All web companies are tracking us. Look for URLs to third-party website found on: Google, Facebook, Twitter and Most mobile apps. Also, some software (e.g., browser) also tracks user actions. They do this for several reasons:
- Security: they can always blocks malicious URL
- Profiling: they know everything you do...

1.2.3 Email phishing

A social engineering technique where an attacker sends a fraudulent ("spoofed") message designed to trick a human victim into revealing sensitive information to the attacker or to deploy malicious software on the victim's infrastructure like ransomware. Two types:

- "general" phishing: mass campaign targeting millions of users. Typically, the user can detect them due to typos and inconsistencies. Relatively easy to detect for email providers that can analyze millions of email boxes.

- spear phishing: targeted attack to a category of users or even a specific single user. Done by a motivated attacker. Hard to automatically detect. Hard to detect for 99 of the users.... me and likely you are in this 99



EXAMPLE OF PHISHING

Primark

Hello ERCOPPA

You are Customer #4644978179 of Primark Rewards and we have been waiting for your confirmation since 15/04/2021 This delievery is for you To activate the delivery ,please Confirm .

Your account information	
Customer:	ERCOPPA
Email:	ercoppa@gmail.com
Reward:	£4975 Primark Gift Card

Continue the delivery

Unsuscribe

This is phishing because they are impersonating a brand.

The name shown is faking the brand:

```
From: "🚨Primark🚨" <FdnXSSMT0Xm72DTJaD@92isr1x8h2m41
1blgepfy.hgu8ygglkj0kogg.fdnxssmt0xm72dtjad.dzoutside.co.com>
```

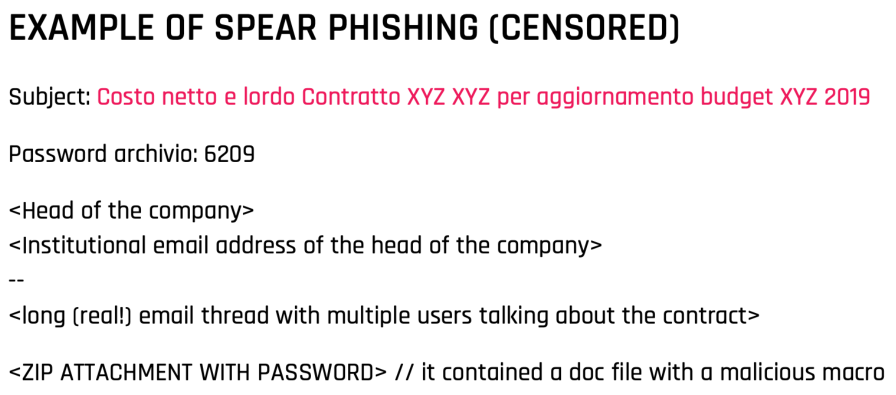
The address is not from Primark but it so long on purpose. Why?

Typo. No real-world big brand will likely mistype their messages.

Bad URLS: if you open them, they show:

Sorry!

The page you were looking for could not be found.



EXAMPLE OF SPEAR PHISHING (CENSORED)

Subject: Costo netto e lordo Contratto XYZ XYZ per aggiornamento budget XYZ 2019

Password archivio: 6209

<Head of the company>
<Institutional email address of the head of the company>
--
<long (real!) email thread with multiple users talking about the contract>

<ZIP ATTACHMENT WITH PASSWORD> // it contained a doc file with a malicious macro

How to (try to) survive?

- SPAM: use a spam filter; most web client have one
- Tracking: do not open links; use anti-tracking features (e.g., Gmail does not show images if the message is marked as spam). Still, "good services" will track you... no matter what you do.
- General phishing: pay attention to the content of the message; use a spam filter.
- Spear Phishing: use your brain; keep in mind that a motivated attacker will find a way to trick you. Keep your software up-to-date!

1.3 Email validation system

We know that there are some pitfalls in the email system. For this reasons, there are other solutions that complements SMTP in order to provide some security features. There are mainly 4 email validation systems which are complementary one to each other so we need all of them:

- Sender Policy Framework (SPF)
  prevents e-mail spam by detecting email spoofing through verification of sender IP addresses

- DomainKeys Identified Mail (DKIM)
  allows to check that incoming mail from a domain is authorized by that domain's administrators and that the email (possibly including attachments) has not been modified during transport

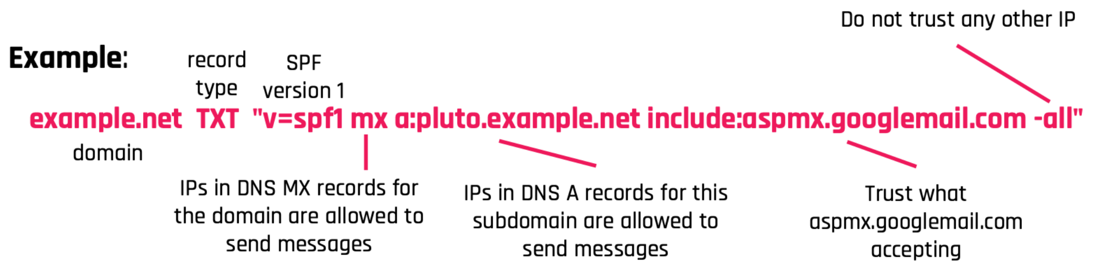
- Domain-based Message Authentication, Reporting and Conformance (DMARC)
  Extends SPF and DKIM with different policies (e.g., how to report spam from a domain?)

- Authenticated Received Chain (ARC)
  A message may traverse a chain of SMTP server: ARC validates the entire chain, even when the message could have been modified (for good reasons).

1.3.1 SPF

One big problems with emails is the fact that the body and the envelope of the email are somewhat separated entities. SPF is a standard to solve this problem from a specific standpoint. In particular, SPF protects the sender address only in the envelope, it doesn't care about the From field in the email header. It allows the administration of the domain through which an email is sent, to define some policies about who could send emails through that server.
The main idea is:

1. A domain publish a DNS TXT record containing the IPs allowed to send messages
2. An SMTP server checks the TXT record to validate the sender's IP
3. The sender's IP is taken by looking at Return-Path (MAIL FROM)


Example:
Do not trust any other IP
record type
SPF version 1
example.net TXT "v=spf1 mx a:pluto.example.net include:aspmx.googlemail.com -all"
domain
IPs in DNS MX records for the domain are allowed to send messages
IPs in DNS A records for this subdomain are allowed to send messages
Trust what aspmx.googlemail.com accepting

Problems:

- SPF only validates the Return-Path but does nothing for From, which is the most frequently spoofed field

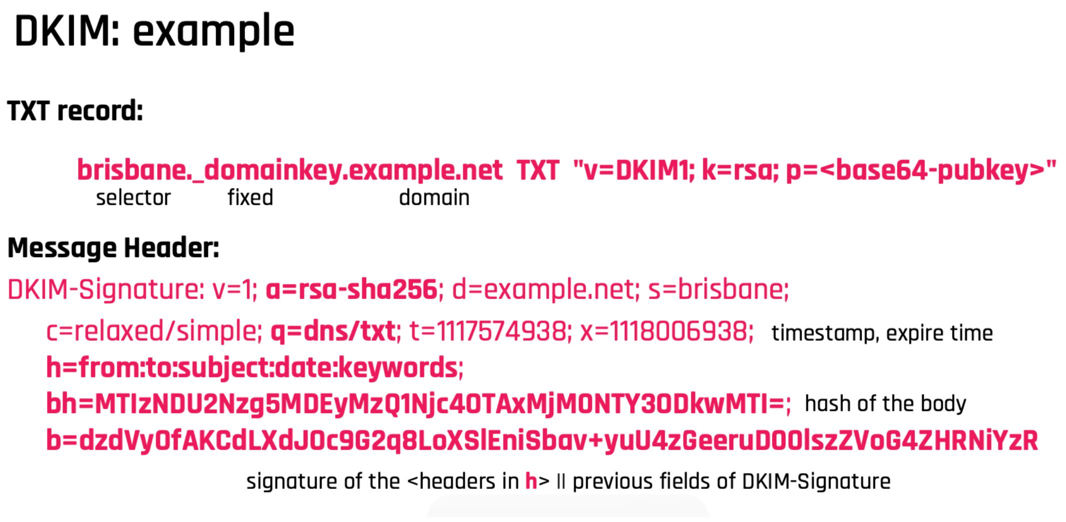
- SPF breaks when a message is forwarded (true only for some forwarding methods): the Sender's IP is not the expected one. However, there could be good reasons to forward emails (e.g., mailing list).

- Just because a message fails SPF, doesn't mean it be will always be blocked from the inbox — it's one of several factors email providers take into account.

- SPF does not authenticate the mail content: what if the content has been altered?

1.3.2 DKIM

It specifies how some parts of the message can be cryptographically signed in order to avoid their content to be tampered by other people (MITM attack or non-intended source creates a fake message with false source information). The main idea: The (server of the) sender signs the message using his private key and this requires to add a DKIM header in the message; different parts of the message could be signed: From is mandatory. When you receive a DKIM signed message, as a recipient you can verify the signature that is contained in the message by querying the domain dns server for the information needed to check that signature. So the idea is that the two will be coherent only if they originates from the same person (domain administrator) or people that the domain administrator allows to use this kind of feature.



DKIM: example

TXT record:

```
brisbane._domainkey.example.net  TXT  "v=DKIM1; k=rsa; p=<base64-pubkey>"
selector  fixed          domain
```

Message Header:

```
DKIM-Signature: v=1; a=rsa-sha256; d=example.net; s=brisbane;
    c=relaxed/simple; q=dns/txt; t=1117574938; x=1118006938;  timestamp, expire time
    h=from:to:subject:date:keywords;
    bh=MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTI=;  hash of the body
    b=dzdVyOfAKCdLXdJOc9G2q8LoXSlEniSbav+yuU4zGeeruD00lszZVoG4ZHRNiYzR
                    signature of the <headers in h> || previous fields of DKIM-Signature
```

Problems:

- Messages could be modified while in transit, potentially invalidating the signature. MITIGATION: DKIM defines CANONICALIZATION rules that allows to tolerate specific (small/cosmetic) changes to some header fields or the body content.
- CANONICALIZATION rules may be not enough in some scenarios, e.g., a mailing list is forwarding a message, modifying the subject and the content
- DKIM does not provide confidentiality
- Domain listed in the DKIM Signature does need to be the same as the one in From: Why? Mailing list scenario: From: is user@gmail.com but the message is sent by another server (malinglist.com) which does not have the private key of gmail.com! It is up to the receiver to accept the message signed by third-party server.

1.3.3 DMARC

DMARC extends SPF and DKIM, allowing an organization to publish a policy that defines its email authentication practices and provides instructions to receiving mail servers for how to enforce them. In other words, DMARC allows a domain to say what to do with a message when DKIM/SPF fails and how to report abuses. The policy is published with a DNS TXT record.

Example:

```
_dmarc.example.com TXT v=DMARC1; p=reject; pct=100;
rua=mailto:aggregate-reports@example.com;ruf=mailto:forensics-reports@example.com
```

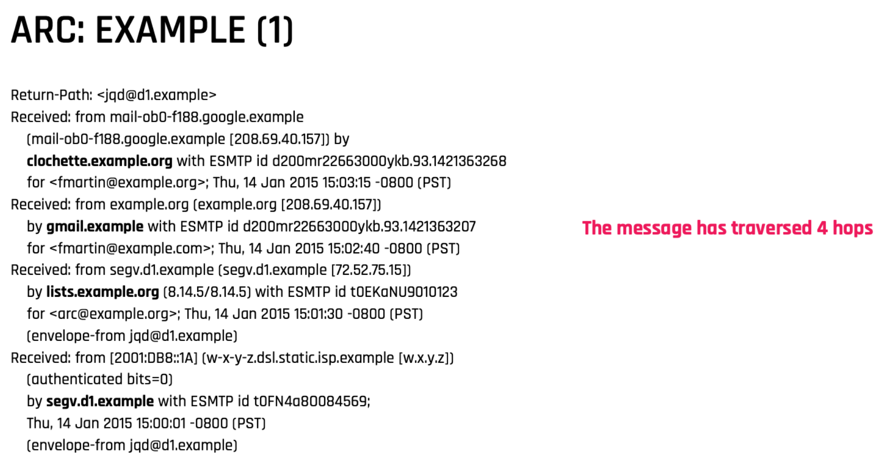
Outcomes from the validation are reported in the mail headers.
How does DMARC works ?
1. The first thing you need to do in order to use DMARC is to publish a policy.
2. When the recipient server receives an incoming email containing the DMARC headers, it performs first of all a dns query to check the policies of the sender. Next, it checks three aspects of the message : checks if the message's DKIM is valid or not, checks if the message come from IP addresses allowed by domain's SPF record and checks if the message headers show proper domain alignment.
3. With this information, the recipient server is ready to apply the sending domain's DMARC policy to decide whether accept, reject or flag the email message.
4. Finally, the recipient server will report the outcome to the sending domain owner. In particular, there are two formats of DMARC reports:

- Aggregate reports: XML documents with statistical data about the messages form a domain. Data includes authentication results and message disposition. Aggregate reports are designed to be machine-readable.
- Forensic reports: individual copies of messages which failed authentication, each enclosed in a full email message using a special format called AFRF. Forensic report can be useful both for troubleshooting a domain's own authentication issues and for identifying malicious domains and web sites.

1.3.4 ARC

DKIM and SPF may break when a message is forwarded or altered by some SMTP servers. ARC provides a way to authenticate the entire chain traversed by a message, while SPF and DKIM authenticate only the original sender. This is done by using additional headers:

- ARC-Authentication-Results: A combination of an instance number and the results of the SPF, DKIM, and DMARC validation
- ARC-Seal: A combination of an instance number, a DKIM-like signature of the previous ARC-Seal headers, and the validity of the prior ARC entries.
- ARC-Message-Signature: A combination of an instance number and a DKIM-like signature of the entire message except for the ARC-Seal headers The basic idea is that each hop in the chain signs the message.



ARC: EXAMPLE (1)

```
Return-Path: <jqd@d1.example>
Received: from mail-ob0-f188.google.example
   (mail-ob0-f188.google.example [208.69.40.157]) by
   clochette.example.org with ESMTP id d200mr22663000ykb.93.1421363268
   for <fmartin@example.org>; Thu, 14 Jan 2015 15:03:15 -0800 (PST)
Received: from example.org (example.org [208.69.40.157])
   by gmail.example with ESMTP id d200mr22663000ykb.93.1421363207
   for <fmartin@example.com>; Thu, 14 Jan 2015 15:02:40 -0800 (PST)
Received: from segv.d1.example (segv.d1.example [72.52.75.15])
   by lists.example.org (8.14.5/8.14.5) with ESMTP id t0EKaNU9010123
   for <arc@example.org>; Thu, 14 Jan 2015 15:01:30 -0800 (PST)
   (envelope-from jqd@d1.example)
Received: from [2001:DB8::1A] (w-x-y-z.dsl.static.isp.example [w.x.y.z])
   (authenticated bits=0)
   by segv.d1.example with ESMTP id t0FN4a80084569;
   Thu, 14 Jan 2015 15:00:01 -0800 (PST)
   (envelope-from jqd@d1.example)
```

The message has traversed 4 hops

1.4 Email Security for the End User

E-mail security needs wrt end-users:

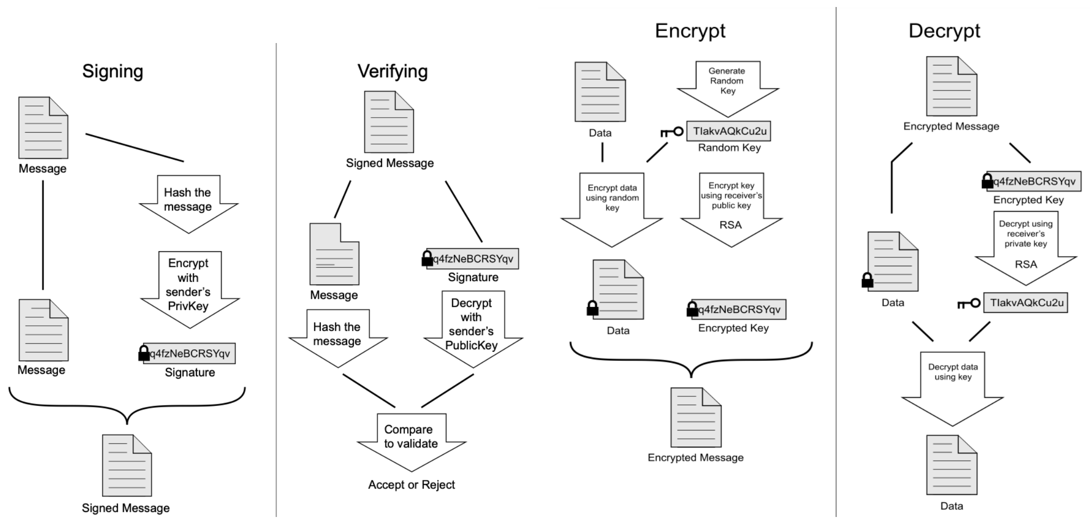
- confidentiality: protection from disclosure
- authentication of sender of message
- message integrity: protection from modification
- non-repudiation of origin: protection from denial by sender

NOTE: Validation systems do not provide these properties to the end-user!

1.4.1 PGP

Pretty Good Privacy (PGP) is an encryption program that provides cryptographic privacy and authentication for data communication. The basic point about PGP is that it was designed by incapsulating the usage of cryptographic primitives inside the email usage process. (open version OpenPGP)

How OpenPGP encryption works visually:



PGP Authentication When the sender creates the message, computes its hash and uses a private key to sign the hash. Next, the message and the signature are put together to form the signed message. On the recipient side, the message signature is decoded using the sender public key and the result is compared with the hash of the message itself. If the two hashes are coherent this means that the message has not been tampered.
PGP Confidentiality First the message is compressed and encrypted by the sender using a symmetric key called session key. Next, the session key is encrypted using RSA with the recipient public key. Next, the encrypted session key and encrypted message are sent together. The recipient takes the encrypted session key and decodes it using its own private key, obtaining the session key, that will be used to decode the original message.

OpenPGP
1. How to embed the signature/encrypted content into a message?
There are several RFCs defining how to handle this problem. Unfortunately, different clients behave in different ways. For instance, given the signature of a message:

- it could be appended as an attachment (e.g., in Thunderbird);
- it could be appended at the end of the message (e.g., in FlowCrypt)

Similar issue wrt encrypted messages. MIME/PGP added specific type/subtypes to MIME to mitigate this issue. Still, it is a bit of mess. The best implementation will adopt one approach but then try to handle also other approaches (e.g., FlowCrypt compatibility list).

2. How to get the public key of other users?
The original idea was Web of Trust: Each user keep a list of (trusted);public keys of other users;He will sign these keys: other users may thus trust these the keys if they trust who is signed them.

Other approaches:

1. The public key is attached to the email
   - what if we want to encrypt? we first have to exchange a message to get the PK
   - should we trust the PK attached to the mail?
   - The idea is that each user has is own way to verify the identity
2. The public key can be fetched from a key server
   - Public servers: is the server verifying the identity of the user upload the PK?
     i. Ubuntu key server: any user can upload a PK, faking the key metadata
     ii. OpenPGP key server: a validation email is sent to the address claimed by the PK
   - Private servers: organizations may track PK of their users. It works but requires special ways of validating the PKs. It cannot scale for all internet users.

PGP: Tutorial:

- Create a new key pair: `gpg --gen-key`

- Export the public key: `gpg --armor --export email@domain.com`
  ```
  -----BEGIN PGP PUBLIC KEY BLOCK-----
  mQGNBGFsTKUBDADAGKYjV8QO/St5OBh8eRZUiwO9fTTy/WLa0gJXaYmzM2qH7Ml3 .....
  lYc6M8/J6HoSNheHYqLsPktWK/zeGOA= =VYi8
  -----END PGP PUBLIC KEY BLOCK-----
  ```
- Import public key of another user: `gpg --import file.asc`
- Import public of another user from a public server: `gpg --recv-keys XYZ`
- Trust imported key: `gpg --sign-key email@example.com`
- Encrypt a file: `gpg --encrypt --armor -r person@email.com file.ext`
- Sign a file: `gpg --sign --armor file.ext` The signature is file.ext.asc
- Encrypt and sign: `gpg --encrypt --sign --armor -r person@email.com file.ext`
- Decrypt and/or check signature: `gpg file.ext.asc`

1.4.2 S/MIME

S/MIME is a widely used protocol for sending and receiving emails in a secure fashion, and its name comes from the fact that it somewhat extend the standard MIME protocol for encapsulating different kind of content in a single document, and it adds MIME types that are used on purpose to handle security mail content. The goal is to provide similar security properties as PGP: authentication, message integrity, non-repudiation of origin (using digital signatures), confidentiality.

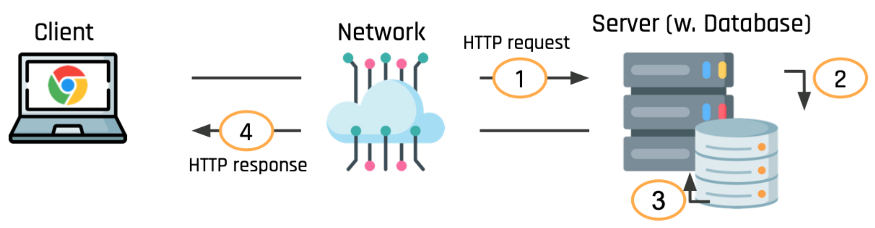
EFAIL (i.e., when email encryption solutions badly fail)
Attacker may access the decrypted content of an email if it contains active content like HTML or JavaScript, or if loading of external content has been enabled in the client.

Suppose the attacker has a copy of encrypted message. E.g., `Content-Type: application/pkcs7-mime; s-mime-typed-envelope-data Content-Transfer-Encoding: base64 <base64-ENCRYPTEDMESSAGEENCRYPTEDMESSAGEENCRYPTEDMESSAGEENCRYPTEDM ESSAGE>`

Now the sends insert the encrypted content inside HTML content. The client of the recipient automatically performs the decryption, replacing the plaintext into the message. If the client is rendering the HTML code, this will trigger a HTTP request for retrieving the image which will leak the secret content to attacker.

2 Web Technologies

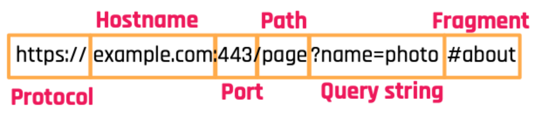
2.1 HTTP



When a user ask for a Typical Web Application:

1. The user request a webpage with dynamically generated content
2. The web application queries the database for user's data
3. The data from the database is used to generate page content
4. The page is rendered by the client's browser

This is done by the use of the URL (Uniform Resource Locator). URLs are identifiers for documents on the Web



- Some elements are optional: port, query string, fragment
- When reserved characters (like space : ? /) need to be used in the URL, they must be URL-encoded:
  - %20 = space
  - %2F = /
  - ...

Example of encoding:

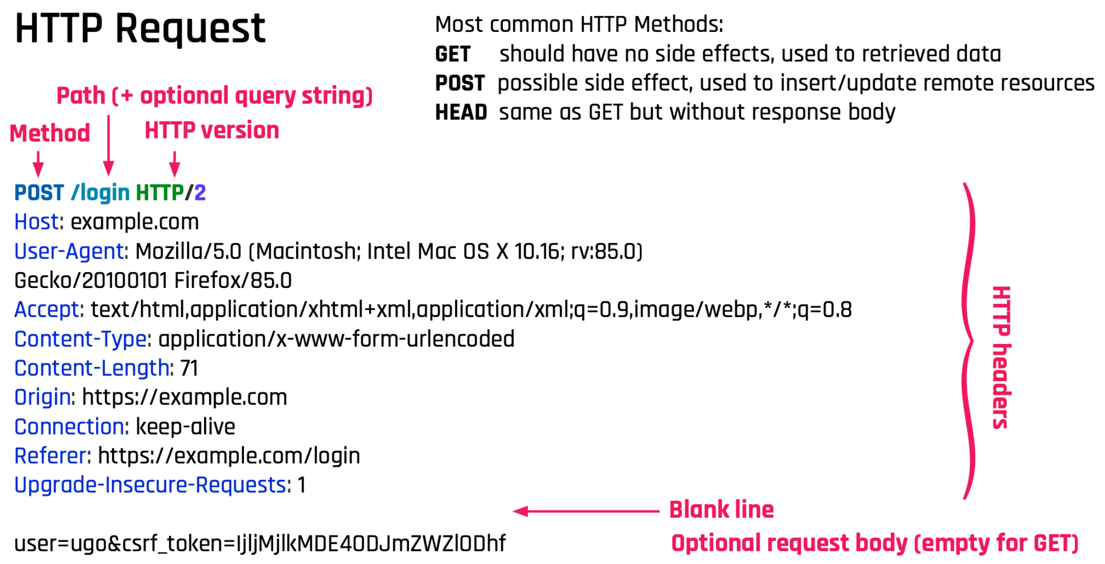
https://example.com/page?name=my%20page

NOTE: For clarity, we will not URL-encode the attack payloads in the next slides

HTTP (Hypertext Transfer Protocol) defines the structure of the communication between client and web server.

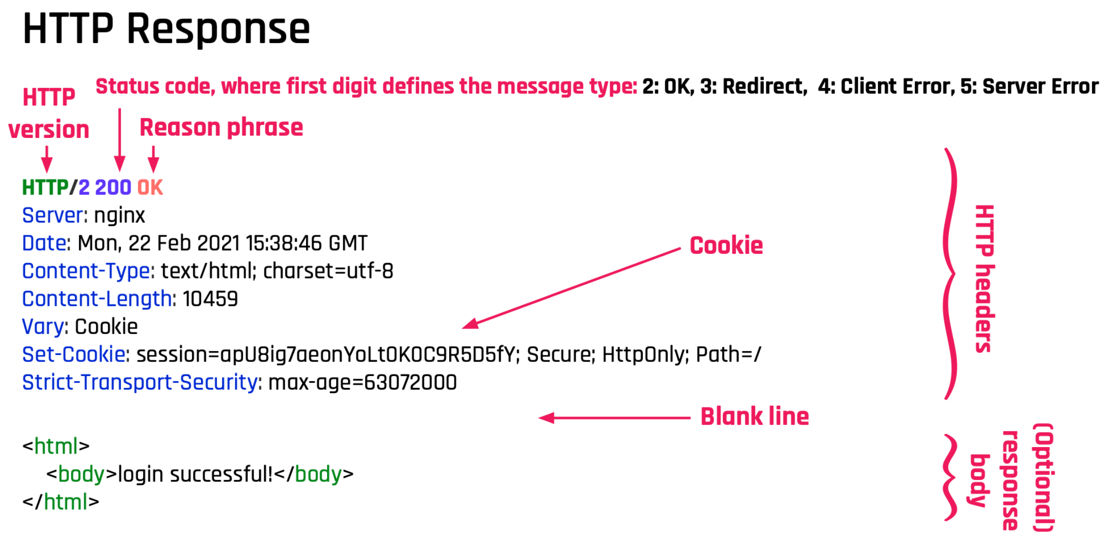
Properties:

- Stateless: different requests are processed independently from each other
  Cookies are used to implement stateful applications on top of HTTP
- Not encrypted: HTTP traffic can be read and modified on the network without the communication parties to notice it
- Default port for HTTP is 80


HTTP Request
Most common HTTP Methods:
GET should have no side effects, used to retrieved data
POST possible side effect, used to insert/update remote resources
HEAD same as GET but without response body
Path (+ optional query string)
Method
HTTP version
HTTP headers
Blank line
Optional request body (empty for GET)

```
POST /login HTTP/2
Host: example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.16; rv:85.0)
Gecko/20100101 Firefox/85.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Content-Type: application/x-www-form-urlencoded
Content-Length: 71
Origin: https://example.com
Connection: keep-alive
Referer: https://example.com/login
Upgrade-Insecure-Requests: 1

user=ugo&csrf_token=IjljMjlkMDE4ODJmZWZlODhf
```


HTTP Response
HTTP version
Status code, where first digit defines the message type: 2: OK, 3: Redirect, 4: Client Error, 5: Server Error
Reason phrase
Cookie
HTTP headers
Blank line
(Optional) response body

```
HTTP/2 200 OK
Server: nginx
Date: Mon, 22 Feb 2021 15:38:46 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 10459
Vary: Cookie
Set-Cookie: session=apU8ig7aeonYoLtOKOC9R5D5fY; Secure; HttpOnly; Path=/
Strict-Transport-Security: max-age=63072000

<html>
  <body>login successful!</body>
</html>
```

2.1.1 HTTPS

HTTPS is the secure variant of HTTP; Essentially, HTTP traffic delivered over a TLS connection. Default port is 443.

Security properties:

- Confidentiality: content of the traffic cannot be inspected as it travels on the network
- Integrity: content of the traffic cannot be modified as it travels on the network
- Authentication: the client can verify that it is communicating with the expected server

2.1.2 HTTP2

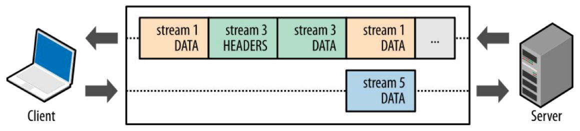
HTTP2 is major revision of the HTTP network protocol. It was derived from the earlier experimental SPDY protocol, originally developed by Google. RFC 7540.
Main goals:

- Provide a negotiation mechanism to select HTTP/1.1, 2.0, or other non-HTTP protocols.
- High-level compatibility with HTTP/1.1
- Decrease latency to improve page load speed:
  - data compression of HTTP headers
  - HTTP/2 Server Push
  - pipelining of requests
  - fixing the head-of-line blocking problem in HTTP 1.x
  - multiplexing multiple requests over a single TCP connection
- Support common existing use cases of HTTP, such as desktop web browsers, mobile web browsers, web APIs, web servers at various scales, proxy servers, etc.

Head-of-line blocking problem in HTTP 1.x - HTTP pipelining is a way to send another request while waiting for the response to a previous request. The idea is to avoid to perform several parallel requests since the setup time for each request could be huge. Problem:
• suppose request B is "queued" (in the pipeline) after request A
• what happens if request A requires a long processing time? Hard to choose how to "queue" the requests. Modern browsers disable pipelining...

HTTP2: some key ideas

- The client can request an upgrade of the connection using the HTTP/1 Request Header. If the server speaks HTTP/2, it sends a "101 Switching" status and from then on it speaks HTTP2 on that connection.
- HTTP2 is a binary protocol, while HTTP is "based" on ASCII. Web applications do not require changes, everything is done by the browser/server.
- The binary protocol allows to efficiently perform multiplexing within one connection:
  - Stream: bidirectional bytes flow within a connection, carrying one or more messages.
  - Message: sequence of frames that map to a logical request or response message.
  - Frame: smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs.



2.1.3 HTTP/3 and QUIC

HTTP/2 addresses head-of-line blocking (HOL) through request multiplexing, which eliminates HOL blocking at the application layer, but HOL still exists at the transport TCP layer! Third revision of HTTP, still in draft. Backward compatible: same request methods, status codes, and message fields. It is based on QUIC, a general-purpose transport layer network protocol designed by Google, which come with two main features:

- Switch from TCP to UDP: this significantly the overhead from the protocol stack, however, now it is up to protocol to recover from transmission errors (a packet is lost). This helps with HOL.

- Integrate into the protocol exchange of setup keys and supported protocols part of the initial handshake process: this reduces overhead for the setup of TLS, which was not optimal in HTTP/2.

2.2 Cookies

HTTP cookies are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or other device by the user's web browser. Cookies are placed on the device used to access a website, and more than one cookie may be placed on a user's device during a session.
Session concept:

- Session data is stored on the server with a unique session ID
- Client attaches the session ID to each request
- Attacker can hijack an (in)active session and impersonate user if session tokens are not properly protected!

Sessions are typically implemented on top of cookies. Cookies set by websites are automatically attached by the browser to subsequent requests to the same website. Cookie attributes (e.g., Domain, Path, Secure, HttpOnly) can be used to customize the cookie behavior. A cookie is identified by the triplet (name, domain, path).

Cookies in practice

- Setting a new cookie (client-side with Javascript):

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path
```

where:

- username is the key (string)
- John Doe is the value (string) associated with the key
- Thu, 18 Dec 2013 12:00:00 UTC is the expiration date. When missing, the cookie expires at the end of the session
- path=/ is the path the cookie belongs to. By default, the cookie belongs to the current page.

- Read all cookies: `let x = document.cookie;`

- Delete a cookie: `document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";` ...you just set the expiration date to a past date.

- Handling cookies in PHP (server side):
`setcookie("user", "John Doe", time() + (86400 * 30), "/");` we set a cookie
`echo $_COOKIE["user"];` we get the value for the cookie
`setcookie("user", "", time() - 3600);` delete the cookie

What are Cookies used for?

- Authentication: The cookie proves that the client previously authenticated correctly
- Personalization: Helps the website recognize the user from a previous visit
- Tracking: Follow the user from site to site; learn his/her browsing behavior, preferences, and so on

Third-party cookies

- a page can host contents coming from other web servers
- cookies that are sent by these servers are named third-party cookies
- there are organizations operating in the advertisement that use third-party cookies for tracking users across different sites, allowing ads consistent to user profile
- This may be a huge privacy concern (we talk about this later in the course)
- Several countries have issued laws on the topic. UE have regulated this matter...

2.3 Storage

Modern browsers allow a web application to store (key, value) pairs on the client:
• Session Storage: kept only for the current session
• Local Storage: permanent across sessions
The key and value can only be strings. The maximum size for the whole storage is typically 5MB, however, it depends from the browser implementation. Web Storage is NOT encrypted: e.g., Firefox uses a SQLite file.

Cookie vs Web Storage

They are similar but different:

- Cookies keep track of data in the client for the server (they are attached to each request!). Web Storage keeps track of data only for the client: the server cannot access it (however, the client may still send its content to the server...).

- Cookies have a maximum size of 4KB. Web Storage is designed with a larger capacity in mind.
- Not always clear what happens if two tabs edit the same cookie at the same time. Web Storage uses DB transactions to deal with concurrent operations.
- Cookies come with very old API, which may lead to some security risks. API for Web Storage were designed later and "should" be better.

Web Storage in practice

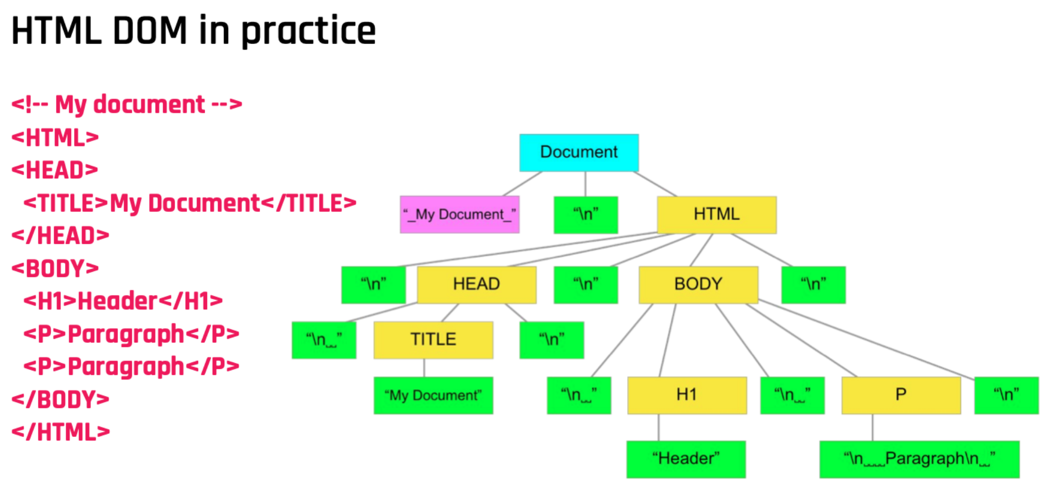
Javascript (client-side):

```
   localStorage.setItem("lastname", "Smith");
   console.log(localStorage.getItem("lastname"));
   localStorage.removeItem("lastname");
```

Same API for sessionStorage.

2.4 DOM

Document Object Model (DOM) is cross-platform and language-independent interface that treats an XML or HTML document as a tree structure. Each node is an object representing a part of the document. Javascript can thus easily modify the a page by modifying the HTML DOM.



HTML DOM in practice

```
<!-- My document -->
<HTML>
<HEAD>
  <TITLE>My Document</TITLE>
</HEAD>
<BODY>
  <H1>Header</H1>
  <P>Paragraph</P>
  <P>Paragraph</P>
</BODY>
</HTML>
```

Javascript (client-side):

• Elements can be retrieved with: `document.getElementById(id)`, `document.getElementsByTagName(name)`, `document.getElementsByClassName(name)`.
E.g.,: `document.getElementById("demo").innerHTML = "Hello World!"; <p id="demo"></p>` `<p id="demo">Hello World!</p>`

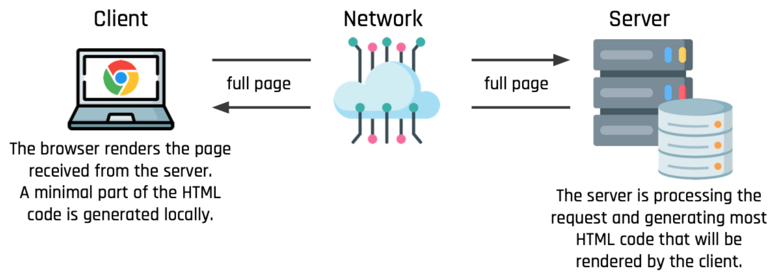
• Given an element, it can be modified using `element.innerHTML` (element content, see example above), `element.<attribute>` (modify an attribute), `element.style.<property>` (modify a CSS property).

• Given an element, we can modify its subtree with: `element.appendChild(element2)`, `element.replaceChild(old, new)`, `element.removeChild(element2)`

- We can create a new element with `document.createElement(<tagname>)`

2.5 Modern Web Applications

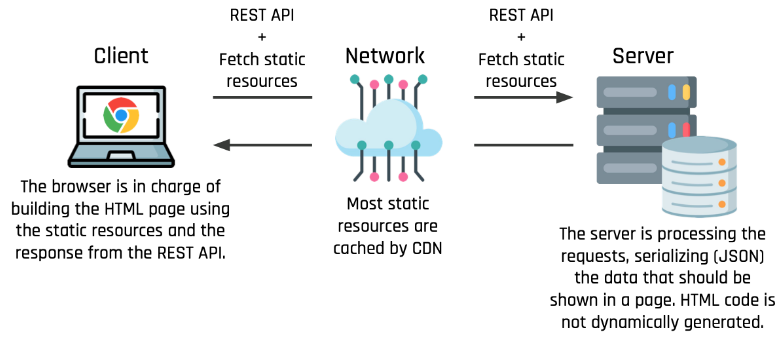
The old but still common approach of web application:


Client
Network
Server
full page
full page
The browser renders the page received from the server. A minimal part of the HTML code is generated locally.
The server is processing the request and generating most HTML code that will be rendered by the client.

Pros:
- browser is doing little work
- Simple logic: most things are done by the server

Cons:
- Hard to scale: large load on the server
- Decoupling:what if want to support a mobile app that has its own way of rendering the content?

Modern approach:


REST API + Fetch static resources
REST API + Fetch static resources
Client
Network
Server
The browser is in charge of building the HTML page using the static resources and the response from the REST API.
Most static resources are cached by CDN
The server is processing the requests, serializing (JSON) the data that should be shown in a page. HTML code is not dynamically generated.

Pros:
- the server is mainly serving static resources and computing the minimal data required for a response
- Clear separation between frontend and backend, making easier to support other client platforms (e.g., mobile app)

Cons:
- Extremely advanced client frameworks

Modern client web frameworks propose the Single-Page Application (SPA) paradigm:

- here is only a single page that is doing all the work. Depending on the URL, the page is built and rendered in different ways.

- when the user clicks something, the page performs a REST request, waits for the response and then renders the new content
- the client framework dynamically modifies the DOM
- there is no need thus reload from scratch the page for each user interaction
- better response time and better user experience
- It is very hard to inspect the code for a human or a bot (e.g., a search engine)

HTTP was not designed for a continuous interactions and push notifications. The browser was designed to make a request and get back the response but in some cases we want an application that sends us notifications. This is hard to implement with the old technologies. That's why a modern browser support was created, which is to say WebSocket, which can be implemented both on client and server side. It provides a persistent connection between client and server and this connection can be close whenever.

2.6 Web Authentication

HTTP defines with RFC 7617 an HTTP transaction for basic authentication:

- The client request a page, e.g.: `GET / HTTP/1.1`
- (Basic)The server replies with: `HTTP/1.1 401 Unauthorized` WWW-Authenticate: `Basic realm=<name-realm>`
- The client gets the username/password from the user with, e.g., a popup and sends: `GET / HTTP/1.1` Authorization: Basic `<BASE64(username:password)>`

  or

- (Digest) The server replies with: `HTTP/1.1 401 Unauthorized` WWW-Authenticate: `Digest realm=<name-realm> nonce=<nonce> opaque=<opaque> algorithm=<algorithm> qop=auth`
  algorithm is a cryptographically secure hash function (default: MD5)
- The client gets the username/password from the user with, e.g., a popup and sends: `GET / HTTP/1.1` Authorization: `texttttDigest realm=<name-realm> nonce=<nonce> user=<user> opaque=<opaque> response=<digest> nc=<counter> cnonce=<cnonce>`
  where digest = `HASH(HASH(<username>:<name-realm>:<passwd>): <nonce>:<counter>: <cnonce>:auth:HASH(GET:/))`

2.7 SSO

OAuth and Single Sign On (SSO) :

- The user has an account on a service provider (Google, Facebook, Linkedin, etc.), also called Identity Provider (IdP)
- The client wants to access a protected resource on a server.
- The server generates a grant code for the client
- The client interacts with IdP, obtaining an access token, which is sent to the server
- The server validates by interacting with IdP, getting also user attributes.

3 Web Security: Attacks and defenses

3.1 The cost of vulnerability

A vulnerability is a weakness which allows an attacker to reduce system's information assurance. A vulnerability is the intersection of three elements:

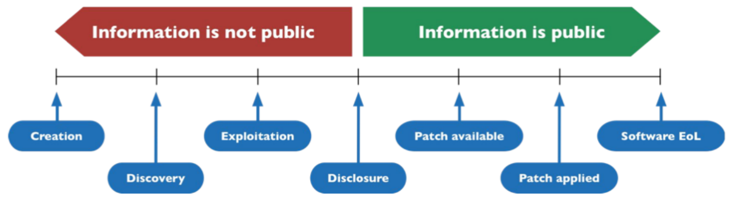
- a system susceptibility or flaw
- attacker access to the flaw
- attacker capability to exploit the flaw

The Causes of vulnerability could be: Bugs, Design defects, Misconfigurations, Aging
Possible fostering factors(effetti collaterali) could be: System complexity, Connectivity,Incompetence.

Vulnerability in software is one of the major reasons for insecurity. Fully secure software is unlikely. 95% of breaches could be prevented by keeping systems up-to-date with patches.

VULNERABILITY LIFE CYCLE

- **CREATION**
- **DISCOVERY**
  - Malicious users
  - Benign users (final users, security firms, researchers, ...)
- **EXPLOITATION**
- **DISCLOSURE**
  - Keep it secret
  - Publicly disclose
  - Sell
- **PATCH**



Responsible disclosure What process should a responsible user follow to disclose the vulnerability?

- No consensus
- Different vendors provides different guidelines for disclosure
- CERT (Computer Emergency Response Team) allows a 45-days grace period. OIS allows a 30-days grace period
- Security firms follow their internal guidelines

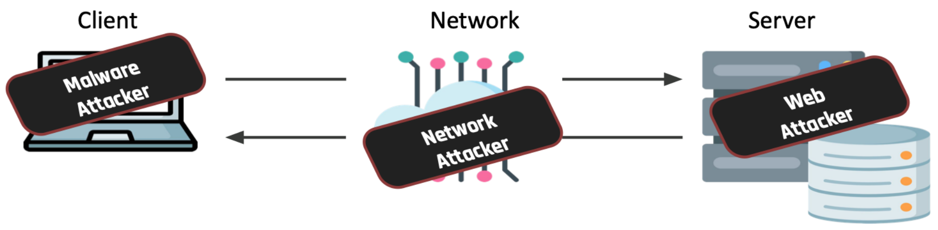
Disclosure policy effects

- Full Vendor Disclosure
  - Promotes secrecy
  - Gives full control of the process to the vendor
- Immediate Public Disclosure
  - Promotes transparency
  - Gives the vendor a strong incentive to fix the problem
  - Allows vulnerable users to take intermediate measures
  - Immediate exposure to risks
- Hybrid Disclosure
  - Promotes both secrecy and transparency

Example: ProxyLogon (2021)
ProxyLogon is a recent vulnerability found on Microsoft Exchange Servers discovered by Orange Tsai (DEVCORE Research Team)
Prerequisites for the attack:
an open HTTPS port (443), i.e., the server is running!
Outcome:
an unauthenticated attacker can execute arbitrary commands on the system (spawn a remote shell, leak all files, ...)

3.2 The Cursed Web

Types of attackers:



Web Attacker

- Attacker controls the domain attacker.com, for which it can acquire a valid TLS certificate. The user visits attacker.com (e.g., because of phishing, search results, click-hijacking, ...)

- Variation “gadget attacker”: an iframe with malicious content included in an otherwise honest webpage visited by the user
- Variation “related-domain attacker”: the attacker controls a related-domain of the target website, e.g., attacker.example.com
- The attacker is a user of a website. The target could be the website or other users. The website should be vulnerable to some attacks.

Network Attackers

- Passive: wireless eavesdropper
- Active: evil Wi-Fi router, DNS poisoning

Malware attacker

- Malicious code executes directly on victim’s computer
- software bugs, malware, ...

The Cursed Web

- Delusive simplicity for creating web apps
- Lack of security awareness
- Time & resource limits during development
- Rapid increase in code complexity

Company’s security focus shifts towards web:
- Security perimeter moves from the network to the application layer
- Web apps intentionally expose functionality to the Internet while being connected to internal servers (e.g., databases)
- Blurred lines between mobile and web apps
  - Web content tightly integrated into mobile apps
  - Unintentional exposure of backend web APIs

3.2.1 OSWAP

The Open Worldwide Application Security Project [7] (OWASP) is an online community that produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security.

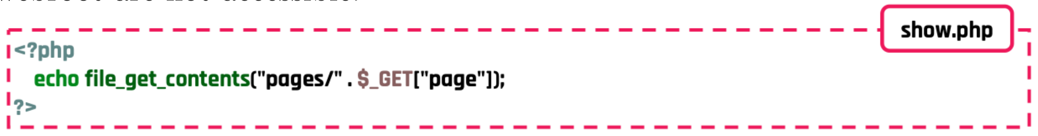
3.3 Threats & Defenses

3.3.1 Path Traversal

Directory traversal (also known as file path traversal) is a web security vulnerability that allows an attacker to read arbitrary files on the server that is running an application. This might include application code and data, credentials for back-end systems, and sensitive operating system files. In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

Example of a Path Traversal Attack

Consider a web server whose webroot is `/var/www/html` (standard location on Linux servers). The webroot is the topmost directory in which the files of a website are stored. Files outside the webroot are not accessible.



```
<?php
  echo file_get_contents("pages/" . $_GET["page"]);
?>
```

The webroot contains the file show.php above and a directory pages containing some text files that can be included by the PHP script

Attacker can "climb up" multiple levels in the directory hierarchy (and exit the webroot) by using ../ (Linux) or .. (Windows) and get access to any file on the web server (sensitive operating system files, TLS keys, etc.). For Example

`/var/www/images/../../../etc/passwd`

The sequence ../ is valid within a file path, and means to step up one level in the directory structure. The three consecutive ../ sequences step up from `/var/www/images/` to the filesystem root, and so the file that is actually read is:

`/etc/passwd`

On Unix-based operating systems, this is a standard file containing details of the users that are registered on the server.

Preventing Path Traversals

- Ideally: don't use user controlled input as (part of) filenames
- In the real world: Validate all user inputs!
  - If possible, allow only a (static) list of file paths
  - Otherwise, compute the canonical path of the required file and ensure it is not outside the webroot (or the expected directory)
- Reduced privileges of web server
  - Restrict access of web server to its own directory
  - Use sandbox environment (chroot jail, SELinux, containers,...) to enforce boundary between web server and the OS

This is a so-called defense-in-depth mechanism: it is a good idea to deploy it, but it should not be the only adopted defense mechanism!

Important directories are

- document root: folder in Web server designated to contain Web pages synonymous: start directory, home directory, web publishing directory, remote root etc. typical names of document root: htdocs, httpdocs, html, public_html, web, www etc.;
- server root: Contains logs & configurations. A few scripts put here their working directory

File permissions

- be aware of permissions given to directories
  - document root, containing HTML documents
  - server root, containing log & configuration files; often CGI scripts run here
  - the Common Gateway Interface (CGI) is a standard (RFC3875) that defines how Web server software can delegate the generation of Web pages to a console application. Such applications are known as CGI scripts; they can be written in any programming language, although scripting languages are often used
- good idea: purposely define user and group for the web server
  - e.g., www & wwwgroup
  - HTML authors should be added to wwwgroup
  - the www should have access only to the right files

Other configurations

Web servers may have additional capabilities, that can increase the risk

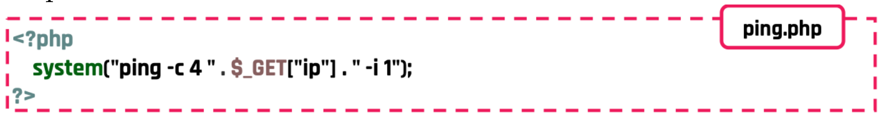
- automatic directory listing: an attacker can see the content of directories... what it we have left some sensitive data (e.g., our editor has left a temp copy of our PHP file?), symbolic links, development logs, source code control directories.
- symbolic link following: it may allow an attacker to access unexpected directories
- server side include (SSI): ".shtml" dynamic pages in the 90s... directives for including other files and executing commands. Still enabled somewhere.
- user maintained directory: Still used in several organization, e.g., example.com/ user

3.3.2 Command & Code Injection

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data. Very often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.

Command Injection Attacks

Most programming languages provide function to execute system commands, e.g., system in PHP Precisely, system starts a new shell (e.g., /bin/bash) which is used to process the command given as parameter to the function.
The page ping.php below uses the system function to ping an IP address provided by the user via the ip variable



```
<?php
  system("ping -c 4 " . $_GET["ip"] . " -i 1");
?>
```

Attack: `GET /ping.php?ip=8.8.8.8; cat /etc/passwd # HTTP/2`
- ";" can be used in almost every shell to combine multiple commands in a single one
- "#" comments the remaining part of the ping command to avoid malformed inputs

Code Injection Attacks

Many interpreted languages provide functions to dynamically evaluate strings as code, e.g., eval in PHP. Idea: I implement an evaluator of numeric expressions and use eval to take advantage of the PHP interpreter! What can go wrong?
`GET /calc.php?expr=2*3 HTTP/2`
Answer: Well, everything!
`GET /calc.php?expr=file_get_contents('/etc/passwd')`

Preventing Code & Command Injection

- NEVER use function like eval that dynamically evaluate strings as code (validation is too error prone here)
- Avoid as much as possible functions that execute system commands and rewrite the code relying on them to use safer alternatives: several programs come with bindings for different languages.
- If you REALLY want to use functions that run system commands, remove properly escape all special characters that break the syntax have a special meaning for the target interpreter (e.g., ; # and so on in bash)
- Reduced privileges of web server
  - Use sandbox environment (e.g., chroot jail, SELinux, containers) to enforce boundary between web server and the OS

3.3.3 SQL Injection

SQL is the declarative language used for querying relational databases. Relational databases build upon the concept of tables (consisting of multiple columns) where the user's data is stored. Sample table users storing data of users registered on a website. On real websites you shouldn't store passwords in cleartext.
Basic SQL Syntax:
Fetch records from a table: `SELECT * FROM users WHERE user='admin' AND password='1f4sdge!';`
Add new records into a table: `INSERT INTO users VALUES ('karl', 's3cr3t', 23);`
Update existing records: `UPDATE users SET age=age+1; DELETE FROM users`
Remove records from a table: `DELETE FROM users WHERE age<25;`
Remove a table from the database: `DROP TABLE users;`

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

Attack:
The attacker uses the following input
user: admin' –
password: whatever
`SELECT * FROM users WHERE user='admin' -- ' AND password='whatever'`
"–" followed by a space starts an inline comment, the part of the query after it is ignored!
In this way the attacker authenticates as the administrator!

`SELECT * FROM users WHERE user='admin' AND password='' OR password LIKE '%';`
"%" matches an arbitrary sequence of characters ,the condition is always satisfied.
In this way The attacker authenticates as the first user in the users table the condition is always satisfied (less control w.r.t. the previous payload)

Stacking Queries
If stacked queries are enabled in the DB configuration, the attacker can perform a variety of attacks harming the integrity of the database:

- Adding a new user to the database: `SELECT * FROM users WHERE user=''; INSERT INTO users (user, password, age) VALUES ('attacker', 'mypwd', 1) -- -' AND password='whatever'`
- Edit the password of the administrator: `SELECT * FROM users WHERE user=''; UPDATE TABLE users SET password='newpwd' WHERE user='admin'-- -' AND password=''`
- Drop the users table from the database: `SELECT * FROM users WHERE user=''; DROP TABLE users -- -' AND password='';`

Pulling Data From Other Tables
Using the injection techniques seen so far, an attacker can dump the contents of the messages table: `'  UNION SELECT user, password FROM users -- -`

```
   SELECT sender, content FROM messages WHERE receiver='attacker' AND content LIKE
'%' UNION SELECT user, password FROM users -- - %'
```
The two SELECT subqueries must return the same number of columns

When the source code of the application is not available and database errors are not displayed on the target website, how can we discover the name of the tables / columns in the database?
We can use the SQL injection to leak the database metadata, which is stored in the information_schema/sqlite_master database!
- information_schema.tables: names of the tables in the various databases of the system
- information_schema.columns: names, types, etc. of the columns of the various tables
`[SQLITE] SELECT * FROM sqlite_master WHERE type='table';`

Second-Order SQL Injection
First-order SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into a SQL query in an unsafe way. In second-order SQL injection (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into a SQL query in an unsafe way.

Suppose that the attacker registers using the following username:
`'; UPDATE TABLE users SET password='newpwd' WHERE user='admin'-- -`
During the login procedure, the username (read from the database) is stored in `$_SESSION["user"]`, which is then used in the query below:
```
SELECT * FROM messages WHERE receiver = ''; UPDATE TABLE users SET password='newpwd'
WHERE user='admin' -- -' AND content LIKE '%%'
```

Blind SQL Injection
Application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors. UNION attacks are ineffective! What can we do instead?
Suppose the query is: `SELECT id FROM users WHERE id = $_GET["id"]`
and that there is no way to show the result of the query. However, the application will react differently depending on the result: e.g., if there is at least one row in the result, then it will show "OK" otherwise "KO!". How can we exploit this behavior?
Assuming that we know: (1) table/column names, (2) a valid id, and (3) the admin username:
```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1,
1) > 'k
```
If "OK" is shown, then we know that the password of the admin starts with a letter greater than

k, otherwise smaller or equal than k. We can do a binary search to identify the exact letter, then move to the next character. NOTE: to leak table/column names, we can exploit a similar technique but on the database metadata!

Another trick is to conditional trigger a SQL error:
`xyz' AND (SELECT CASE WHEN (Username = 'Administrator' AND SUBSTRING(Password, 1, 1) > 'm') THEN 1/0 ELSE 'a' END FROM Users)='a`
The final query generates a division by zero (1/0) when the condition that we want to test is false, otherwise it is valid ('a'='a').

Another trick is to introduce a delay when a desired condition is verified:
`'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator' AND SUBSTRING(Password, 1, 1) > 'm') = 1 WAITFOR DELAY '0:0:10'--`
The final query takes more than 10 seconds when the condition that we want to test is true.

Other tricks:
• Check the number of columns for a table: e.g., test 4 columns `(SELECT 1, 2, 3, 4) = (SELECT * FROM 'Table_Name')`
Fatal error when the table does not have 4 columns

• [SQLITE] Leak scheme of a table: `SELECT REPLACE(sql, X'0A', "") FROM sqlite_master WHERE type != 'meta' AND sql NOT NULL AND name NOT LIKE 'sqlite_%'`
`AND name ='Table_Name';`

Preventing SQL Injections

- Use prepared statements: they allow to embed untrusted parameters in a query, while ensuring that its syntactical structure is preserved
- Rely on whitelisting approaches ONLY when prepared statements cannot be used (e.g., when the input is the name of the table to be used in FROM or ORDER BY) e.g., allow only safe characters like letters, digits and underscore
- Restrict access to sensitive tables with database permissions (defense-in-depth protection). However, this cannot be done when these tables are required to implement the functionalities of the web application

3.3.4 SSRF

What is a Server-Side Request?
A server may need to perform some internal/external connections to serve the client request. For instance:
• the "ping service" example in the previous slides: DNS request, ICMP request
• authentication via, e.g., Single-Sign On (SSO): a request to the identify provider
• captcha verification: a request to validate the user response
• REST API:

- the backend may use some external services
- the backend may use some internal services

Server-side request forgery (also known as SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make requests to an unintended location. In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems, potentially leaking sensitive data such as authorization credentials.
When the request or some of its aspects can be manipulated by the user then an attacker may be able to forge an "unexpected" request:

- the end target (URI) of the request is under the control of the user:
  `FROM https://auth.service/ TO https://attacker.com` [control the response]
  `FROM https://auth.service/ TO https://local.ip/` [map/access the internal network]
  `FROM https://auth.service/secret-token TO https://attacker.com/secret-token` [data leak]

- the data sent are under the control of the user:
  `FROM https://social.com/newpost=AAA TO https://social.com/newpost=‘cat /etc/passwd‘`

Other consequences of SSRF:

- Some internal services may be sometimes accessible without any authentication when the request is coming from the internal network. Via SSRF, we may thus able to freely access the service. Examples: admin panels,databases,log handlers, infrastructure monitors

- Cloud providers may expose sensitive metadata through specific internal hosts. E.g., AWS exposes instance metadata at http://169.254.169.254 which may leak sensitive information.

Blind SSRF
In some cases, the server will not provide any explicit feedback about the request: e.g., the response of the server-side request is not shown to the user and we are unable to perform external connections. We can still perform nasty things with a blind SSRF:

- get a feedback by looking at the time required for the server-side request: e.g., the server-side request may take a different time depending on the internal host (invalid or not) and port (open or closed) that we are testing.

- blindly execute requests/actions in the internal lan

Preventing SSRF

- whitelist approach: requests are made only towards specific hosts. Hard to do when we want to allow connections even to unexpected hosts.

- isolated host: the host performing the request should be isolated from the rest of the network and should not have access to any sensitive data

3.4 Javascript and Same Origin Policy (SOP)

The basic Execution Model of the Browser is based on the fact that each browser window/tab/frame loads content and renders pages (Processes HTML, stylesheets and scripts to display the page) and reacts to events (via JavaScript). Some commands of Javascript are:
- User actions: OnClick, OnMouseover, ...
- Rendering: OnLoad, OnUnload, ...
- Timing: setTimeout, clearTimeout, ...

Scripts can be embedded in a page in multiple ways:

- Inlined in the page: `<script>alert("Hello World!");</script>`
- Stored in external files: `<script type="text/javascript" src="foo.js"></script>`
- Specified as event handlers: `<a href="http://www.bar.com" onmouseover="alert('hi');">`
- Pseudo-URLs in links: `<a href="javascript:alert('You clicked');">Click me</a>`

JavaScript can interact with the HTML page and the browser through the DOM and the BOM. The Browser Object Model (BOM) is a browser-specific convention referring to all the objects exposed by the web browser. Unlike the Document Object Model, there is no standard for implementation and no strict definition, so browser vendors are free to implement the BOM in any way they wish. JavaScript can dynamically modify the DOM, e.g., to add a new item to the list or to add an event handler to the items in the list.

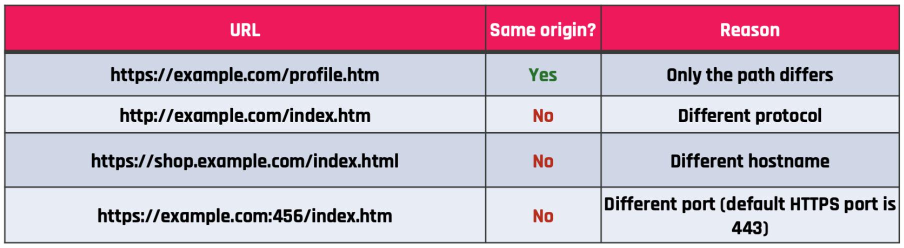
Browser Sandbox
Goal: safely execute JavaScript code provided by a remote website by enforcing isolation from resources provided by other websites. No direct file access and limited access to OS,network, browser data, content that came from other websites.

SOP The same-origin policy (SOP) is baseline security policy that restricts how a document or script loaded by one origin can interact with a resource from another origin. An origin is defined as the triplet (protocol, domain, port). Two URLs have the same origin if the protocol, port (if specified), and host are the same for both. Scripts running on a page hosted at a certain origin can access only resources from the same origin:

- access (read / write) to DOM of other frames
- access (read / write) to the cookie jar (different concept of origin, we will see it later) and local/session storage
- access (read) to the body of a network response

Some aspects are not subject to SOP like:inclusion of resources (images, scripts, ...),form submission and sending requests (e.g., via the fetch API).

Example: https://example.com/index.htm



URL	Same origin?	Reason
https://example.com/profile.htm	Yes	Only the path differs
http://example.com/index.htm	No	Different protocol
https://shop.example.com/index.html	No	Different hostname
https://example.com:456/index.htm	No	Different port (default HTTPS port is 443)

Despite being a fundamental web security mechanism, there is no formal definition of SOP! Full policy of current browsers is complex:
- Evolved via "penetrate-and-patch"
- Different features evolved in slightly different policies
- A recent study evaluated 10 different browsers and discovered that browsers behave differently in 23% of the tests (focus only on DOM access)

DNS Rebinding
DNS rebinding is a method of manipulating resolution of domain names that is commonly used as a form of computer attack. In this attack, a malicious web page causes visitors to run a client-side script that attacks machines elsewhere on the network. In theory, the same-origin policy prevents this from happening: client-side scripts are only allowed to access content on the same host that served the script. Comparing domain names is an essential part of enforcing this policy, so DNS rebinding circumvents this protection by abusing the Domain Name System (DNS).
This attack can be used to breach a private network by causing the victim's web browser to access computers at private IP addresses and return the results to the attacker.

Mitigations against DNS Rebinding

- DNS Pinning:
  - Browsers could lock the IP address to the value received in the first DNS response
  - Compatibility issue with some dynamic DNS uses, load balancing, etc.
- Web servers can reject HTTP requests with an unrecognized Host headers: Default catchall virtual hosts in the web server configuration should be avoided

3.5 JSON with Padding (JSON-P)

Sometimes cross-origin read is desired... Developers came up with JSON-P, a technique exploiting the fact that script inclusion is not subject to the SOP.

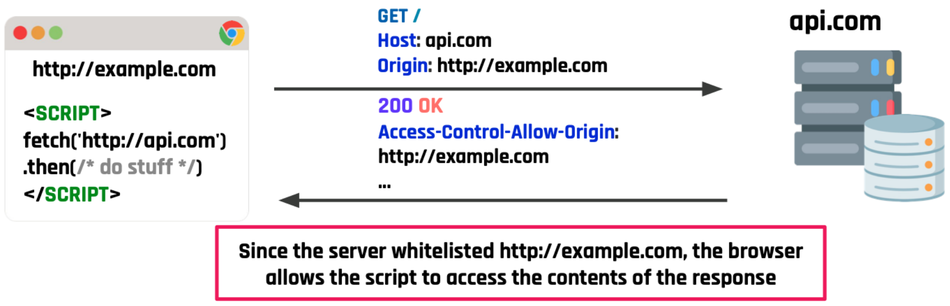
Issues with JSON-P:

- Only GET requests can be performed
- Endpoint could validate Referer but this may be forged or missing

- Requires complete trust of the third-party host
  - The third-party is allowed to execute scripts within the importing page
  - The importing origin cannot perform any validation of the included script
- JSON-P should not be used anymore!

3.6 CORS

Sometimes it is desirable to allow JavaScript to access the content of cross-site resources. Cross-Origin Resource Sharing (CORS) provides a controlled way to relax the SOP. JavaScript can access the response content if the Origin header in the request matches the Access-Control-Allow-Origin header in the response (or the latter has value *)


http://example.com
GET /
Host: api.com
Origin: http://example.com
api.com
200 OK
Access-Control-Allow-Origin:
http://example.com
...
Since the server whitelisted http://example.com, the browser allows the script to access the contents of the response

```
<SCRIPT>
fetch('http://api.com')
.then(/* do stuff */)
</SCRIPT>
```

CORS Headers:

- Request headers (used in pre-flight request):
  - Access-Control-Request-Method: the HTTP method that will be used in the actual request
  - Access-Control-Request-Headers: list of custom HTTP headers that will be sent in the actual request
- Response headers:
  - Access-Control-Allow-Origin: used to whitelist origins, allowed values are null, * or an origin (value * cannot be used if Access-Control-Allow-Credentials is specified)
  - Access-Control-Allow-Methods: list of allowed HTTP methods
  - Access-Control-Allow-Headers: list of custom HTTP headers allowed
  - Access-Control-Expose-Headers: list of response HTTP headers that will be available to JS
  - Access-Control-Allow-Credentials: used when the request includes client credentials
  - Access-Control-Max-Age: used for caching pre-flight requests

Pitfalls in CORS Configurations

- Two different CORS specifications existed until recently:
  - W3C: allows a list of origins in Access-Control-Allow-Origin
  - Fetch API: allows a single origin in Access-Control-Allow-Origin
  - Browsers implement CORS from the Fetch API (and the W3C one is now deprecated)
- Browsers implementations complicate CORS configuration: Server-side applications need custom code to validate allowed origins rather than just providing a static header with all the whitelisted origins

Pitfall #1 - Broken Origin Validation:
Snippet of nginx configuration setting the CORS header:
```
if ($http_origin - "http://(example.com|foo.com)") {
add_header "Access-Control-Allow-Origin" $http_origin; }
```

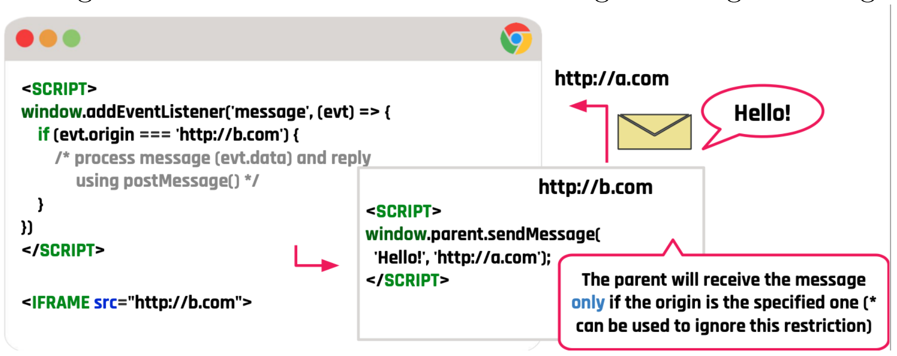
Allowed origins:
- http://example.com
- http://foo.com
- http://example.com.evil.com

Pitfall #2 - The null origin
The Access-Control-Allow-Origin header may specify the null value. Browsers may send the Origin header with a null value in particular conditions:
- Cross-site redirects
- Requests using the file: protocol
- Sandboxed cross-origin requests

An attacker can forge requests with the null Origin header by performing cross-origin requests from a sandboxed iframe

3.7 Client-side Messaging

postMessage is a web API that enables cross-origin message exchanges between windows.


```
<SCRIPT>
window.addEventListener('message', (evt) => {
  if (evt.origin === 'http://b.com') {
    /* process message (evt.data) and reply
       using postMessage() */
  }
})
</SCRIPT>

<IFRAME src="http://b.com">
```
```
<SCRIPT>
window.parent.sendMessage(
 'Hello!', 'http://a.com');
</SCRIPT>
```
http://a.com
Hello!
http://b.com
The parent will receive the message only if the origin is the specified one (* can be used to ignore this restriction)

Message handlers should validate the origin field of incoming messages in order to communicate only with the desired origins. Failures to do so may result in security vulnerabilities, e.g., when the received message is evaluated as a script or unsafely embedded into a page. A recent study found 377 vulnerable message handlers on the top 100k sites. Some of these lacked origin checking, others were implementing it in the wrong way (e.g., substring match)

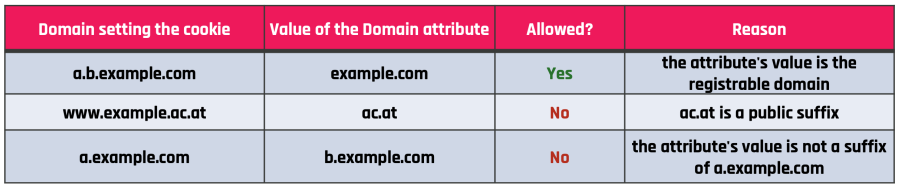
3.7.1 Cookies

Scope of Cookie

The domain attribute widens the scope of a cookie to all subdomains. If one subdomain is compromised, such cookies will be leaked to unauthorized parties. To restrict the scope of a cookie to the domain that set it, the domain attribute must not be specified

The Domain Attribute

If the attribute is not set, the cookie is attached only to requests to the domain who set the cookie. If the attribute is set, the cookie is attached to requests to the specified domain and all its subdomains. The value can be any suffix of the domain of the page setting the cookie, up to the registrable domain.
A related-domain attacker can set cookies that are sent to the target website!



Domain setting the cookie	Value of the Domain attribute	Allowed?	Reason
a.b.example.com	example.com	Yes	the attribute's value is the registrable domain
www.example.ac.at	ac.at	No	ac.at is a public suffix
a.example.com	b.example.com	No	the attribute's value is not a suffix of a.example.com

Cookie Attributes

The Path attribute can be used to restrict the scope of a cookie, i.e., the cookie is attached to a request only if its path is a prefix of the path of the request's URL. Useful, e.g.: `example.com/ userA vs example.com/ userB`
- If the attribute is not set, the path is that of the page setting the cookie
- If the attribute is set, there are no restrictions on its value
-If the Secure attribute is set, the cookie will be attached only to HTTPS requests (confidentiality). Since recently, browsers prevent Secure cookies to be set (or overwritten) by HTTP requests (integrity).
- If the HttpOnly attribute is set, JavaScript cannot read the value of the cookie via document.cookie:

• No integrity is provided: a script can overflow the cookie jar, so that older cookies are deleted, and then set a new cookie with the desired value

• Prevents the theft of sensitive cookies (e.g., those storing session identifiers) in case of XSS vulnerabilities

Max-Age or Expires define when the cookie expires. When both are unset, the cookie is deleted when the browser is closed. When Max-Age is a negative number or Expires is a date in the past, the cookie is deleted from the cookie jar. If both are specified, Max-Age has precedence

The SameSite Attributes

A request is cross-site if the domain of the target URL of the request and that of the page triggering

the request do not share the same registrable domain
- A request from a.example.com to b.example.com is same-site (the registrable domain is example.com)
- A request from example.com to bank.com is cross-site

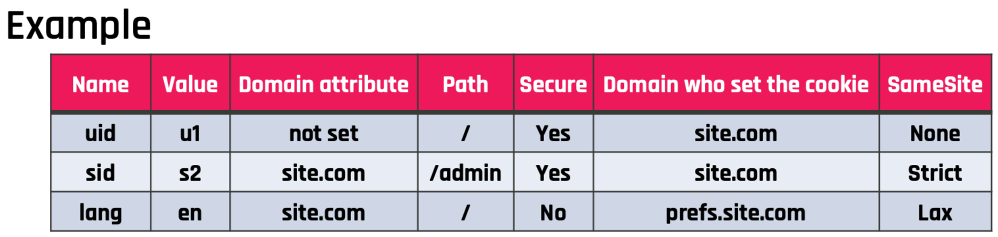
The SameSite attribute controls whether the cookie should be attached to cross-site requests:

- Strict: the cookie is never attached to cross-site requests CSRF Protection
- Lax: the cookie is sent even in case of cross-domain requests, but then there must be a change in top-level navigation (user realizes it!)
- None: the cookie is always attached to all cross-site requests

Recent Changes to Cookies (Feb. 2020)
- SameSite = Lax by default: Cookies that do not explicitly set the SameSite attribute are treated as if they specified SameSite = Lax;
Before February 2020, these cookies were treated as if they set SameSite = None
- SameSite = None implies Secure : Cookies with attribute SameSite set to None are discarded by the browser if the Secure attribute is specified as well

SOP for Reading Cookies
A cookie is attached to a request towards the URL u if the following constraints are satisfied:

- if the Domain attribute is set, it is a domain-suffix of the hostname of u, otherwise the hostname of u must be equal to the domain of the page who set the cookie
- the Path attribute is a prefix of the path of u
- if the Secure attribute is set, the protocol of u must be HTTPS
- if the request is cross-site, take into account the requirements imposed by the SameSite attribute



Example

Name	Value	Domain attribute	Path	Secure	Domain who set the cookie	SameSite
uid	u1	not set	/	Yes	site.com	None
sid	s2	site.com	/admin	Yes	site.com	Strict
lang	en	site.com	/	No	prefs.site.com	Lax

Which cookies are attached to a cross-site request from https://www.example.com (triggered by the user clicking on a link, changing the top-level navigation context) to:
http://site.com/ -> lang=en
https://site.com/ -> uid=u1;lang=en
https://site.com/admin/ -> uid=u1;lang=en
https://a.site.com/admin/ -> lang=en

Cookies Protocol Issues

The Cookie header, which contains the cookies attached by the browser, only contains the name and the value of the attached cookies

- the server cannot know if the cookie was set over a secure connection
- the server does not know which domain has set the received cookie
- RFC 2109 has an option for including domain, path in the Cookie header, but it is not supported by any browser (and is now deprecated)

Cookie Jar Overflow

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, older cookies are deleted
- Attackers can thus overflow the cookie jar to "overwrite" HttpOnly cookies or to bypass cookie tossing protection on servers that block requests with multiple cookies having the same name

Cookie Prefixes

Cookie prefixes have been proposed to provide to the server more information on the security guarantees provided by cookies:
- __Secure-: if a cookie name has this prefix, it will only be accepted by the browser if it is marked as Secure
- __Host-: If a cookie name has this prefix, it will only be accepted by the browser if it is marked Secure, does not include a Domain attribute, and has the Path attribute set to /

Recap: SOP vs JSON-P vs CORS

- SOP defines the concept of origin (protocol, hostname, port)
  - it restricts DOM access and networks requests to the origin
  - it does not restrict content inclusion (e.g., scripts)
  - it is more relaxed when dealing with cookies
- How to perform a network cross-origin request (A => B)?
  - JSON-P (hack): the idea is to include a script, whose content is dynamically generated by the B, that when executed by A will send data to one function from A. In practice, A can thus receive data from B.
  - CORS (proper way): the browser will allow A to read the response from B only if B explicitly whitelists A using a CORS header

Recap: cookies

Main attributes:

- Domain: when set, cookie can be accessed even by related domains. Hence, a cookie could be accessible by different origins from the same registrable domain.
- Path: when set, access to cookie is restricted based on the path
- Secure: when set, cookie is set only when using HTTPS
- HttpOnly: when set, javascript code cannot access cookie
- SameSite: whether a cookie is appended in case a cross-site request. Notice that site means a registrable domain (a.foo.com and b.foo.com have the same "site": foo.com)

Recap: problems of cookies

- Cookie Jar is organized based on (name, domain, path). However, its handling is browser specific:
  - Cookie tossing attack: subdomain A sets cookie X that can be accessed by subdomain B. What if B had already a cookie called X? The browser will define an "order" on the cookies and the attacker may exploit it.
  - Cookie Jar Overflow: given a cookie X with attribute HttpOnly, javascript code should not be able to change its value. However, an attacker may generate a large number of cookies, forcing the browser to discard the cookie X. Then, the attacker can arbitrarily set X using javascript code.
- When a cookie is sent in a request, only name=value is sent. Hence, the server is not aware and cannot verify the cookie attributes in the client (e.g., to reject the cookie if httpOnly is false).
- Cross-site requests may leak the content of a cookie when SameSite is not set correctly. We will some examples later on when we consider XSS.

3.7.2 CSRF

Cross-site request forgery (also known as CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.
Assume that the victim is authenticated on the target website; the attack works as follows:

- the victim visits the attacker's website
- the page provided by the attacker triggers a request towards the victim website, e.g.:
  - forms automatically submitted via JavaScript for CSRF on POST requests
  - loading of an image for CSRF on GET requests
- the cookie identifying the session is automatically attached by the browser!

Example:
1. `[->]` POST /login
Host: bank.com user=alice&pass=mypwd
`[<-]` 200 OK
Set-Cookie: session=XYZ;
Secure; HttpOnly

2.`[->]` GET / Host: evil.com
`[<-]`200 OK
```
<HTML>
<BODY>
<IMG src="https://bank.com/ ?act=attacker&amt=3000"> </BODY>
</HTML>
```

3. `[->]` GET /transfer?act=attacker
&amt=3000
Host: bank.com
Cookie: session=XYZ
`[<-]` Transfer executed!

Problem: bank.com cannot distinguish legitimate requests from those triggered by a third-party website

CSRF Defenses: Anti-CSRF Tokens

- Synchronizer token pattern (forms)

  - A secret, randomly generated string is embedded by the web application in all HTML forms as a hidden input field
  - Upon form submission, the application checks whether the request contains the token: if not, the sensitive operation is not performed
    `<INPUT type="hidden" value="ak34F9dmAvp">`

- Cookie-to-header token (JavaScript)

  - A cookie with a randomly generated token is set upon the first visit of the web application
  - JavaScript reads the value of the cookie and embeds it into a custom HTTP header
  - The server verifies that the custom header is present and its values matches that of the cookie
    Set-Cookie: `__Host-CSRF_token=aen4GjH9b3s; Path=/; Secure`

- Possible design choices for the generation of CSRF tokens:

  - Refreshed at every page load: limits the timeframe in which a leaked token (e.g., via XSS) is valid

- Generated once on session setup: improves the usability of the previous solution, which may break when navigating the same site on multiple tabs

- To be effective, CSRF tokens must be bound to a specific user session: Otherwise an attacker may obtain a valid CSRF token from his account on the target website and use it to perform the attack!

Most modern web frameworks use anti-CSRF tokens by default!

Referer Validation attack
Some applications make use of the HTTP Referer header to attempt to defend against CSRF attacks, normally by verifying that the request originated from the application's own domain. This is generally less effective than CSRF token validation.
Sometimes the Referer header is suppressed:

- Stripped by the organization's network filter
- Stripped by the local machine
- Stripped by the browser for HTTPS → HTTP transitions
- User preferences in browser

Different types of validation:
- Lenient: requests without the header are accepted. May leave room for vulnerabilities!
- Strict: requests without the header are rejected Could block legitimate requests

CSRF Mitigations

The SameSite cookie attribute provides an effective mitigation against CSRF attacks:

- Since the recent browser updates (SameSite = Lax by default), sites are protected by default against classic web attackers
- No protection is given against related-domain attackers: requests from the attacker domain to the target website are same-site!

Fetch Metadata is another approach to mitigate CSRF vulnerabilities. The idea underlying Fetch Metadata is to provide to the server (via HTTP headers) some information about the context in which a request is generated. The server can use this information to drop suspicious requests (e.g., legitimate bank transfers are not triggered by image tags). Can be used to mitigate also XSSI, clickjacking, etc.
Currently supported only by Chromium-based browsers
For privacy reasons, headers are sent only over HTTPS.

Fetch Metadata Headers

- `Sec-Fetch-Dest`: specifies the destination of the request, i.e., how the response contents will be processed (image, script, stylesheet, document, ...)

- `Sec-Fetch-Mode`: the mode of the request, as specified by the Fetch API
  Is it a resource request subject to CORS?
  Is it a document navigation request?
- `Sec-Fetch-Site`: specifies the relation between the origin of the request initiator and that of the target, taking redirects into account
  Is it a cross-site, same-site or same-origin request?
- `Sec-Fetch-User`: sent when the request is triggered by a user action

Sample Policies:
Resource isolation policy, mitigates CSRF, XSSI, timing side-channels: `Sec-Fetch-Site == 'cross-site' AND (Sec-Fetch-Mode != 'navigate'/'nested-navigate' OR method NOT IN [GET, HEAD])`

Navigation isolation policy, mitigates clickjacking and reflected XSS: `Sec-Fetch-Site == 'cross-site' AND Sec-Fetch-Mode == 'navigate'/'nested-navigate'`

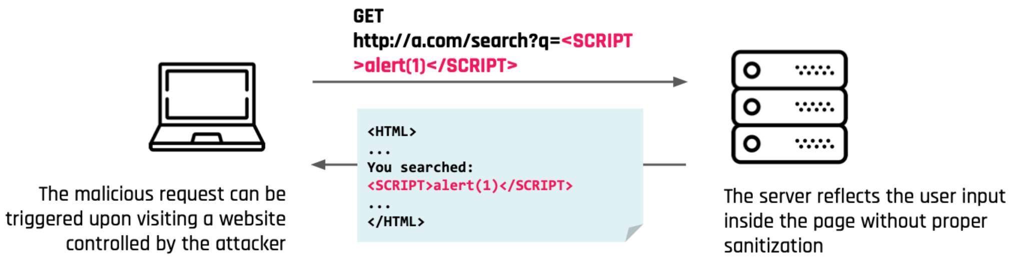
3.7.3 XSS

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other. Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data. If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.
Root cause of the problem: improper sanitization of user inputs before they are embedded into the page

Type of XSS vulnerabilities:

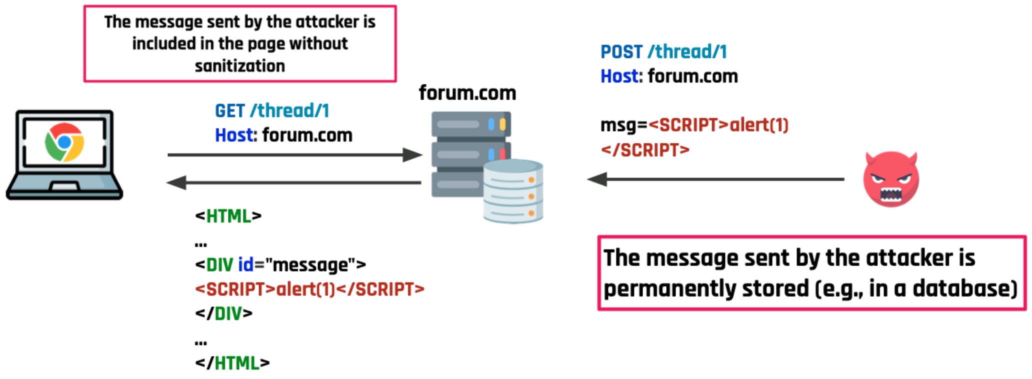
- Reflected: data from the request is embedded by the server into the web page
- Stored: the payload is permanently stored on the server-side, e.g., in the database of the web application
- DOM-based: the payload is unsafely embedded into the web page on the browser-side

Reflected XSS The website includes data from the incoming HTTP request into the web page without proper sanitization.
User is tricked into visiting an honest website with an URL prepared by the attacker (phishing email, redirect from the attacker's website, ...).
Script can manipulate website contents (DOM) to show bogus information, leak sensitive data


GET
http://a.com/search?q=<SCRIPT
>alert(1)</SCRIPT>
The malicious request can be triggered upon visiting a website controlled by the attacker
The server reflects the user input inside the page without proper sanitization

```
<HTML>
...
You searched:
<SCRIPT>alert(1)</SCRIPT>
...
</HTML>
```

Stored XSS Website receives and stores data from an untrusted source. Lots of websites serving user-generated content: social sites, blogs, forums, wikis, e.g., attacker embeds script as part of comment in a forum. When the visitor loads the page, website displays the content and visitor's browser executes the script. No interaction with the attacker is required!


The message sent by the attacker is included in the page without sanitization
forum.com
GET /thread/1
Host: forum.com
POST /thread/1
Host: forum.com
msg=<SCRIPT>alert(1)
</SCRIPT>
The message sent by the attacker is permanently stored (e.g., in a database)

```
<HTML>
...
<DIV id="message">
<SCRIPT>alert(1)</SCRIPT>
</DIV>
...
</HTML>
```

DOM-based XSS In DOM-based XSS, data from an attacker-controllable source (e.g., the URL) is entered into a sensitive sink or browser APIs without proper sanitization. Sensitive sinks are properties / functions that allow to modify the HTML of the web page (e.g., to add new scripts) or the execution of JavaScript code (e.g., eval).
The injection of dangerous code is performed by vulnerable JS code: the server may never see the attacker's payload! and server-side detection techniques do not work!

DOM XSS occurs when one of injection sinks in DOM or other browser APIs is called with user-controlled data.
For example, consider this snippet that loads a stylesheet for a given template

```
const templateId = location.hash.match(/tplid=([^;&]*)/)[1];
// ...
document.head.innerHTML += `<link rel="stylesheet" href="./templates/${templateId}/style.css">`
```

- This code introduces DOM XSS by linking the attacker-controlled source (`location.hash`) with the injection sink (`innerHTML`)

```
https://example.com#tplid="><img src=x onerror=alert(1)>
```

XSS Prevention

- Any user input must be preprocessed before it is used inside the page: HTML special characters must be properly encoded before being inserted into the page
  Depending on the position in which the input is inserted, different encodings or filtering may be required (e.g., within an HTML attribute vs inside a <div> block)
- Don't do escaping manually!
  Use a good escaping library:
  - OWASP ESAPI (Enterprise Security API)
  - Microsoft's AntiXSS
  - DOMPurify (client-side)

  Rely on templating libraries which provide escaping features:
  - Smarty and Mustache in PHP, Jinja in Python, ...
- Against DOM-based XSS, use Trusted Types!

Caveats with Filters
Suppose that a XSS filter removes the string <script from the input parameters:
`<script src="..." becomes src="..."`
`<scr<scriptipt src="..." becomes <script src="..."`
Need to loop and reapply until nothing found. However, ¡script may not be necessary to perform an XSS:
`<A href="INJECTION_HERE">` +
`#" onclick="alert(1)` =
`<A href="#" onclick="alert(1)">`

3.7.4 CSP

Content Security Policy (CSP) is a browser security mechanism that in origin aims to mitigate XSS and some other attacks. Now it is used for many different purposes like: restrict framing capabilities, blocking mixed contents, restrict targets of form submissions, restrict URLs to which the document can start navigations (via forms, links, etc.). It works by restricting the resources (such as scripts and images) that a page can load and restricting whether a page can be framed by other pages. To enable CSP, a response needs to include an HTTP response header called Content-Security-Policy with a value containing the policy. The policy itself consists of one or more directives, separated by semicolons.

CSP Directives
CSP allows fine-grained filtering of resources depending on their type

- `font-src, frame-src, img-src, media-src, script-src, style-src, ...`
- `default-src` is applied when a more specific directive is missing

A list of values can be specified for each directive:

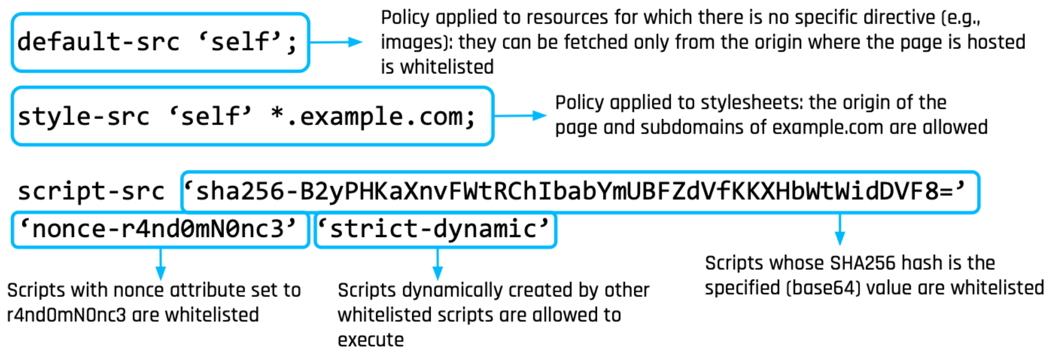
- hosts (with * as wildcard): http://a.com, b.com, *.c.com, d.com:443, *
- schemes: http:, https:, data:
- 'self' whitelists the origin from which the page is fetched
- 'none' whitelists no URL

The following directive values are specific to scripts / stylesheets:

- 'unsafe-inline' whitelists all inline style directives / scripts (including event handlers, JavaScript URIs, ...)
- 'unsafe-eval' allows the usage of dynamic code evaluation functions (e.g., eval)
- 'nonce-<value>' whitelists the elements having the specified value in the nonce attribute
- 'sha256-<value>', 'sha384-<value>', 'sha512-<value>' whitelist the elements having the specified hash value (which is encoded in base64)
- 'unsafe-hashes' is used together with a hash directive value to whitelist inline event handlers
- 'strict-dynamic' allows the execution of scripts dynamically created by other scripts

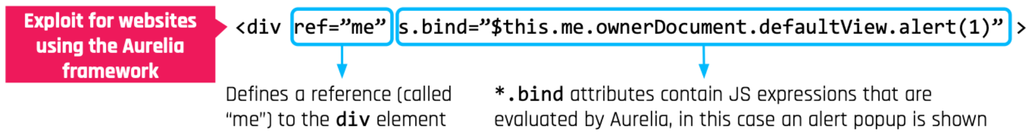
Some values are incompatible with others:

- when nonces are used, 'unsafe-inline' is ignored
- when 'strict-dynamic' is used, whitelists and 'unsafe-inline' are ignored


default-src 'self';
Policy applied to resources for which there is no specific directive (e.g., images): they can be fetched only from the origin where the page is hosted is whitelisted
style-src 'self' *.example.com;
Policy applied to stylesheets: the origin of the page and subdomains of example.com are allowed
script-src 'sha256-B2yPHKaXnvFWtRChIbabYmUBFZdVfKKXHbWtWidDVF8='
'nonce-r4nd0mN0nc3' 'strict-dynamic'
Scripts with nonce attribute set to r4nd0mN0nc3 are whitelisted
Scripts dynamically created by other whitelisted scripts are allowed to execute
Scripts whose SHA256 hash is the specified (base64) value are whitelisted

Bypassing CSP with Code Reuse Attacks

Many websites use very popular (and complex) JS frameworks. Examples include AngularJS, React, Vue.js, Aurelia, ...
These frameworks contain script gadgets, pieces of JavaScript that react to the presence of specifically formed DOM elements. Idea: abuse scripts gadgets to obtain code execution by injecting benign-looking HTML elements. In a nutshell, we're bringing code-reuse attacks (like return-to-libc in binaries) to the Web.


Exploit for websites using the Aurelia framework
Defines a reference (called "me") to the **div** element
***.bind** attributes contain JS expressions that are evaluated by Aurelia, in this case an alert popup is shown

```
<div ref=”me” s.bind=”$this.me.ownerDocument.defaultView.alert(1)” >
```

3.7.5 Trusted Types

New API pushed by Google to obliterate DOM XSS.
Idea:

- Lock down dangerous injection sinks so that they cannot be called with strings
- Interaction with those functions is only permitted via special (trusted) typed objects
- Those objects can be created only inside a Trusted Type Policy (JS code part of the web application)
- Policies are enforcement by setting the trusted-types special value in the CSP response header
- Ideally, TT-enforced applications are “secure by default” and the only code that could introduce a DOM XSS vulnerability is in the policies

Identified >60 different injection sinks
3 possible Trusted Types:

- TrustedHTML strings that can be confidently inserted into injection sinks and rendered as HTML
- TrustedScript ... into injection sinks that might execute code
- TrustedScriptURL ... into injection sinks that will parse them as URLs of an external script resource

Possible pitfalls

- Non DOM XSS could lead to a bypass of the policy restrictions
- Sanitisation is left as an exercise to the policy writers
- Policies are custom JavaScript code that may depend on the global state
- New bypass vectors/injection sinks yet to be discovered?

3.7.6 HSTS

SSL stripping attacks are a type of cyber attack in which hackers downgrade a web connection from the more secure HTTPS to the less secure HTTP.
The HTTP Strict-Transport-Security response header (often abbreviated as HSTS) informs browsers that the site should only be accessed using HTTPS, and that any future attempts to access it using HTTP should automatically be converted to HTTPS.
Attackers can still perform SSL stripping the first time a site is visited... Browsers ship with a preload list of websites which are known to support HTTPS.
Bypassing HSTS with NTP

- The Network Time Protocol (NTP) is used to synchronize the clock between different machines over a network

- Most operating systems use NTP without authentication, being thus vulnerable to network attacks.
  Some OS accept any time contained in the response (e.g., Ubuntu, Fedora)
  Other OS impose constraints on the time difference (at most 15 48 hours on Windows, big time differences allowed only once in macOS)

- Idea: make the HSTS policies expire by forging NTP responses containing a time far ahead in the future

4 Web Tracking

Goals of web tracking: determining web users interests or modeling users behavior aiming at enabling the most appropriate advertising, may violate the user privacy, may be used for unethical purposes

User tracking Several ways of uniquely identifying users:

1. Using cookie across websites
2. Client fingerprinting
3. Network tracking

4.1 Tracking Cookies

3rd party cookies enable user tracking in many different ways, aggressive tracking pages may use even 50 different techniques. Websites which browser interacts with.

Difference between First-Party Cookie and Third-Party Cookie:

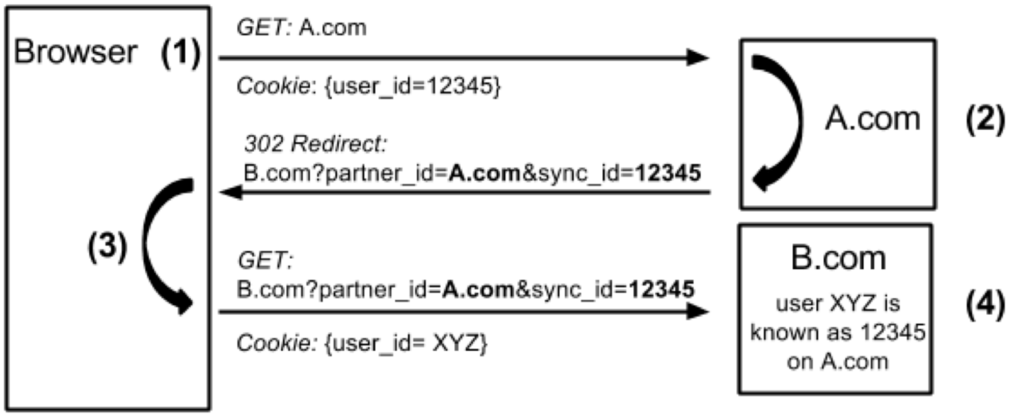
- Setting and reading the cookie:
  - FPC: Can be set by the publisher's web server or any JavaScript loaded on the website
  - TPC: Can be set by a third-party server via code loaded on the publisher's website
- Availability:
  - FPC: A first-party cookie is available only via domain that create it
  - TPC: A third-party cookie is available on any website that the third-party server's code
- Browser supporting, Blocking and Deletion:
  - FPC: Supported by all browser and can be blocked and delete by the user, but doing it may provide a bad user experience.
  -TPC: Supporting by all browser, but many are now blocking the creation of third-party cookie by default. Many users also delete third-party cookie on regular basis.

Tracking based on 3rd party cookies

1. connect to site A to get index.html
2. index.html asks to download image img.png from site B $\neq$ A
3. connect to B to get img.png
   - B knows the referer (if no https)
   - B sends one or more cookies with sensitive information

A referrer is the URL of a previous item which led to the current request. For an image is generally the page on which it is to be displayed. The referrer is an optional field of the HTTP request, websites log referrers as part of their attempt to track their users. Some web browsers allow to disable the sending of referrer information. Referrer removal may break the functioning of a webpage.

Cookies syncing
Cookie syncing is a web browser feature that allows signed-in users to synchronize the browsing data from one device with the same user on another.


Browser (1)
GET: A.com
Cookie: {user_id=12345}
A.com (2)
302 Redirect:
B.com?partner_id=A.com&sync_id=12345
(3)
GET:
B.com?partner_id=A.com&sync_id=12345
Cookie: {user_id= XYZ}
B.com
user XYZ is known as 12345 on A.com
(4)

Suppose the user clears the cookie for B.com and then visits other websites which also perform cookie syncing with B.com, obtaining a new ID, e.g., 678910. If the user visits again A.com, after the sync, B.com can understand that 12345 is the same as 678910.

4.2 Client Fingerprinting

Fingerprinting is a tracking and identification method websites use to associate individual browsing sessions with one site visitor.

4.3 TOR

Internet is designed as a public network. Wi-Fi access points, network routers see all traffic that passes through them. Routing information is public:
-IP packet headers identify source and destination
- Even a passive observer can easily figure out who is talking to whom
Encryption does not hide identities, Encryption hides payload, but not routing information. Even IP-level encryption (tunnel-mode IPsec/ESP) reveals IP addresses of IPsec gateways.

Anonymity = the person is not identifiable within a set of subjects

- You cannot be anonymous by yourself!
- Big difference between anonymity and confidentiality
- Hide your activities among others' similar activities

Unlinkability of action and identity. For example, sender and his email are no more related after adversary's observations than they were before.

Unobservability (hard to achieve): Adversary can't even tell whether someone is using a particular system and/or protocol.

Attacks on Anonymity

- Passive traffic analysis: Infer from network traffic who is talking to whom
- Active traffic analysis: Inject packets or put a timing signature on packet flow
- Compromise of network nodes:
  - Attacker may compromise some routers
  It is not obvious which nodes have been compromised:
  - Attacker may be passively logging traffic
  - Better not to trust any individual router
  - Can assume that some fraction of routers is good, but don't know which

Tor
Tor project deployed onion routing network. The goal of onion routing was to have a way to use the internet with as much privacy as possible, and the idea was to route traffic through multiple servers and encrypt it each step of the way. Individuals use Tor to keep websites from tracking them, or to connect to those internet services blocked by their local Internet providers. Tor's hidden services let users publish web sites and other services without needing to reveal the location of the site. Data content cannot be accessed in the network. However, routing information and other metadata are still available!

Components of Tor:

- Client: the user of the Tor network
- Server: the target TCP applications such as web servers
- Tor (onion) router: the special proxy relays the application data
- Directory server: servers holding Tor router information

Tor operations:how to works

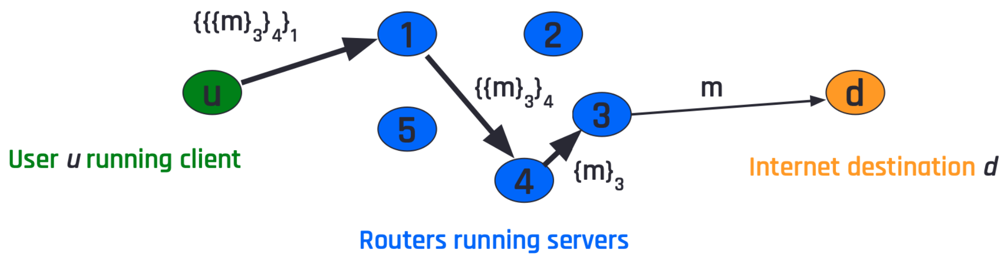
1. Alice's Tor client obtains a list of Tor nodes from directory server
2. Alice's Tor client picks a random path to destination server. (internal links are encrypted, instead the last link, or the links between destinations are in clear.)
3. If at a later time, the user visit another site, Alice's Tor client selects a second random path.

Onion Routing A circuit is built incrementally one hop by one hop.
Onion-like encryption:

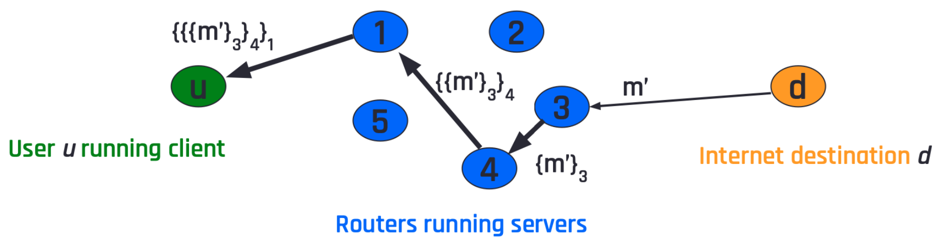
- Alice negotiates an AES key with each router

- Messages are divided into equal sized cells
- Each router knows only its predecessor and successor
- Only the Exit router (OR3) can see the message, however it does not know where the message is from

Example:



1. u creates l-hop circuit through routers
2. u opens a stream in the circuit to d
3. Data are exchanged



4. Stream is closed.

Tor Circuit Setup: 1. Client proxy establish a symmetric session key and circuit with relay node #1
2. Client proxy extends the circuit by establishing a symmetric session key with relay node #2 (Tunnel through relay node #1)
3. Client proxy extends the circuit by establishing a symmetric session key with relay node #3. (Tunnel through relay nodes #1 and #2)
4. Client applications connect and communicate over the established Tor circuit. Datagrams decrypted and re-encrypted at each link.

Hidden Services
Goal: deploy a server on the Internet that anyone can connect to without knowing where it is or who runs it. Accessible from anywhere and resistant to censorship, denial of service, physical attack. Network address of the server is hidden, thus can't find the physical server. Location-hidden services allow Bob to offer a TCP service without revealing his IP address. Tor accommodates receiver anonymity by allowing location hidden services.
Design goals for location hidden services:

- Access Control: filtering incoming requests (against flooding)
- Robustness: maintain a long-term pseudonymous identity (ability to migrate service across Ors)
- Smear-resistance: social attacker should not be able to "frame" a rendezvous router by offering an illegal location-hidden service and making observers believe the router created that service
- Application transparency: same unmodified application

Location hidden service leverage rendezvous points.

Setting up a hidden service

1. Bob generates a long-term public key pair to identify his service
2. Bob chooses some introduction points, and advertises them on the lookup service, also providing his public key
3. Bob builds a circuit to each of his introduction points, and tells them to wait for requests.
4. Alice learns about Bob's service out of band, retrieves the details of Bob's service from the lookup service
5. Alice chooses an OR as the rendezvous point (RP) for her connection to Bob's service. She builds a circuit to the RP, and gives it a randomly chosen "rendezvous cookie" to recognize Bob.
6. Alice opens an anonymous stream to one of Bob's introduction points, and gives it a message (encrypted with Bob's public key) telling it about herself, her RP and rendezvous cookie, and the start of a DH handshake. The introduction point sends the message to Bob.
7. If Bob wants to talk to Alice, he builds a circuit to Alice's RP and sends the rendezvous cookie, the second half of the DH handshake, and a hash of the session key they now share.
8. The RP connects the two circuits. Note that RP can't recognize Alice, Bob, or data they transmit.
9. Alice sends a relay begin cell along the circuit. It arrives at Bob's OP, which connects to Bob's webserver.
10. An anonymous stream has been established, and Alice and Bob communicate as normal.