

Email

explained from first principles

This [article](#) and its [code](#) were first published on 7 May 2021 and [last modified](#) on 13 May 2021. If you like the article, please share it with your friends on [social media](#) or support me with a [donation](#). You can also join the [discussion on Reddit](#), [download the article](#) as a PDF, or [use Google Translate](#) to read this article in your native language.

If you are visiting this website for the first time, then please first read the [front page](#), where I explain the intention of this blog and how to best make use of it. As far as your privacy is concerned, all data entered on this page is stored locally in your browser unless noted otherwise. While I researched the content on this page thoroughly, you take or omit actions based on it at your own risk. In no event shall I as the author be liable for any damages arising from information or advice on this website or on referenced websites.

▼ Preface

Being one of the oldest services on the Internet, email has been with us for decades and will remain with us for at least another decade. Even though email plays an important role in everyday life, most people know very little about how it works. Before we roll up our sleeves and change this, here are a few things that you should know:

- This article covers all aspects of modern email. As a result, it became really long. While later chapters do build on earlier ones, you can start reading wherever you want and fill your knowledge gaps as you go.
- This article is structured as follows: After clarifying some user-facing [concepts](#), we'll look at the technical [architecture](#) of email and the roles of the various [entities](#). We'll then study the [protocols](#) used by these entities to communicate with one another and the [format](#) of the transmitted messages. Once we understand how email works, we can discuss its [privacy](#) and [security issues](#) and examine how some of the security issues are being [fixed](#) by more recent standards.
- Among many other things, you will learn in this article [why mail clients use outgoing mail servers](#), [why SMTP is used for the submission and the relay of messages](#), [how mail loops are prevented](#), and [how you should configure your custom domains](#).
- Even if you're not interested in email, this article can teach you a lot about Internet protocols and IT security. For example, it covers [Implicit and Explicit TLS](#), [password-based authentication mechanisms with hash functions](#), [replay attacks](#), [encryption mechanisms](#), and [channel bindings](#); [internationalized domain names with Punycode encoding](#), [Unicode normalization](#), [case folding](#), and [homograph attacks](#); [transport security with DANE and HSTS](#); and [end-to-end security with S/MIME and PGP](#).
- If you haven't done so already, read the [article about the Internet](#) first. This article assumes that you're familiar with the following acronyms and the concepts behind them: [RFC](#), [IP](#), [TCP](#), [TLS](#), [DNS](#), and [DNSSEC](#).
- This article contains 29 tools. To make it easier to play around with them, I've published them on a [separate page](#) as well.
- This article focuses on how modern email works, not on how you set up your own email infrastructure. If you want to do that, [Mail-in-a-Box](#) seems like a good place to start.
- During my research for this article, I made [responsible disclosures](#) to [Gandi](#), [Microsoft](#), and [Mozilla Thunderbird](#). I also submitted [quite a few RFC errata](#).

▼ Terminology

[Email](#), which also used to be written as e-mail, stands for electronic mail. Since the term *electronic mail* applies to any mail that is transferred electronically, it also encompasses [fax](#), [SMS](#), and other systems. For this reason, I use only the short form *email* in this article and always mean the decentralized system to transfer messages over the Internet as documented in numerous [RFCs](#). The term *email* doesn't appear in the [original RFC](#) and many RFCs just use *mail* or *(Internet) message* instead. In ordinary language, *email* refers both to the system of standards and to individual messages transmitted via these standards. While the English language would allow us to distinguish between the two usages by capitalizing the former but not the latter, I've never seen anyone doing this. Even though I'm tempted to pioneer the proper use of grammar here, I'd rather save my [artistic license](#) for other things. ([Proper nouns](#) refer to a single entity, whereas common nouns refer to a class of entities. Only proper nouns are capitalized in English. For example, *Earth* with a capital E refers to the planet we live on, whereas *earth* with a lowercase E refers to the soil in which plants grow.) Note that this is in contrast to [Internet](#), which is commonly capitalized because there is only one Internet: You're either connected to *the* Internet or not. Unfortunately, the Internet becomes increasingly fragmented along country borders due to legal reasons, such as copyright licenses, and political reasons, such as censorship. Therefore, we might have to degrade *Internet* to a common noun soon.

Concepts

Before diving into the technical aspects of email, let's first look at email from the perspective of its users.

Message

The purpose of email is to send messages over the Internet. A message is a recorded piece of information which is delivered asynchronously from a sender to one or several recipients. Asynchronous communication means that a message can be consumed at an arbitrary point after it has been produced, rather than having to interact with the sender concurrently. A message can be transmitted with a physical object, such as a letter, or with a physical signal, such as an acoustic or electromagnetic wave. While humans have delivered messages in the form of objects for millennia with couriers and pigeons, it's only since the invention of the optical telegraph in the late 18th century and the invention of the electrical telegraph in the middle of the 19th century that we can signal arbitrary messages over long distances. The fundamental principle of communication stayed the same over all those years: You can either start a new conversation or continue an existing one by replying to a previous message.

Mailbox

A mailbox is a box for incoming mail (also called an inbox), into which everyone can deposit messages but ideally only the intended recipient can retrieve them. In some countries, the privacy of such messages is legally protected by the secrecy of correspondence.

Provider

There are three things that set email apart from the traditional postal system, which is sometimes also referred to as snail mail:

1. Email conveys digital data, whereas a letter is an analog item. The former is much more useful for further processing.
2. Email enables instant global delivery at a marginal cost of zero. The only fee you pay is for your access to the Internet.
3. Mailboxes for email are provided and operated by companies, which are called email service providers. While you could operate your own server since email is an open and decentralized system, this is rarely done in practice for reasons we discuss later on.

▼ Which are the most popular email service providers?

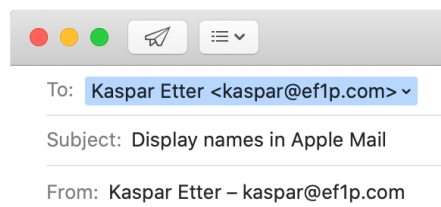
It is estimated that around half of the human population uses email, with an average of 1.75 active accounts per user. (Please treat all the numbers in this box with caution. They were surprisingly hard to come by, with the sources being scattered and not necessarily trustworthy. Additionally, the following numbers were reported in different years, which distorts the actual market share of these companies.) In the Western world, the consumer market is dominated by Google with their Gmail service, which has 1.5 billion active users. In China, the biggest player is Tencent QQ with 900 million active accounts. Outlook by Microsoft has 400 million active users, which is followed by Yahoo! Mail with 225 million active users. Apple's iCloud has 850 million users but it's not known how many of those use its email functionality.

Address

Email addresses are used to identify the sender and the recipient(s) of a message. They consist of a username followed by the @symbol and a domain name. The domain name allows the sender to first determine and then connect to the mail server of each recipient. The username allows the mail server to determine the mailbox to which a message should be delivered. The hierarchical Domain Name System ensures that the domain name is unique, whereas the email service provider has to ensure that the name of each user is unique within its domain. There doesn't have to be a one-to-one correspondence between addresses and mailboxes: A mailbox can be identified by several addresses, and an email sent to a single address can be delivered to multiple mailboxes.

▼ Display name

Email protocols accept an optional display name in most places where an email address is expected. The format for this is `Display Name <user@example.com>` according to RFC 5322. Mail clients display this name to the user as follows:



How Apple Mail shows the display name in the To and From fields
– if you have Smart Addresses disabled, which you totally should.

This feature seems totally benign, but, as we will see later on, it has serious privacy and security implications.

▼ The @ symbol

While most of us know the [@ symbol](#) exclusively from email addresses and social media to tag another user, it has been used for centuries in commerce. In Spanish and Portuguese, it denoted a [custom unit of weight](#). In English, it came to mean *at the rate of* similar to the French *à*. The @ symbol was already included in the first edition of the [ASCII character set](#) in 1963, years before the symbol was first used to designate the [network host](#) in a predecessor of today's email in 1971.

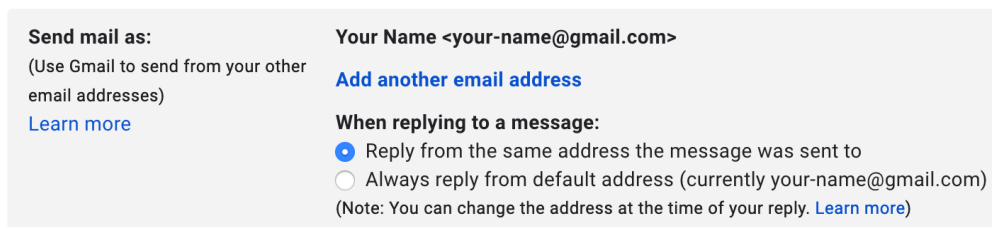
▼ Normalization

In [the standard](#), the part before the @ symbol is called the [local part](#) of an email address. The interpretation of the local part is completely up to the receiving mail system specified after the @ symbol and you shouldn't make any assumptions about the recipient's address as a sender. In particular, implementations [must preserve the case](#) of the letters in the local part, but mail servers are encouraged to deliver messages case-independently. In other words, it is recommended but not mandatory that mail servers treat John.Smith and john.smith as the same user. Some email service providers go further than this: Gmail, for example, [removes all dots](#) from the local part of an address when determining the mailbox to deliver a message to. This means that emails addressed to john.smith@gmail.com and johnsmith@gmail.com are received by the same user – who also gets all messages for j.o.h.n.s.m.i.t.h@gmail.com. The process of transforming data to its canonical form is called [normalization](#).

▼ Subaddressing

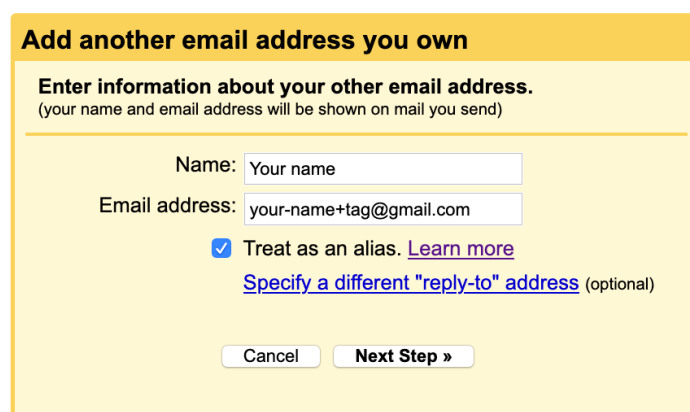
Many email service providers support a technique known as [subaddressing](#) as part of their address normalization. By restricting the character set for usernames more than the standard demands, an email service provider can designate a special character, which is valid according to the standard but not in its set for usernames, to split the [local part](#) into two. The part before this special character is used to determine the recipient of a message. The part after this special character is a tag that the user chose when they shared their address. [Gmail](#) and [Microsoft Exchange](#) support subaddressing with a plus. For example, emails to user+tag@gmail.com will be delivered to user@gmail.com.

If you reply to an email that you received at a subaddress with a plus, Gmail still uses your main address in the From field, unfortunately. In order to send emails (including replies) from a subaddress, you have to add it [in the settings](#):



The screenshot shows the 'Send mail as' settings in Gmail. It includes a section for 'Your Name <your-name@gmail.com>' with a link to 'Add another email address'. Below this is a section 'When replying to a message:' with two radio button options: 'Reply from the same address the message was sent to' (selected) and 'Always reply from default address (currently your-name@gmail.com)'. A note at the bottom states: '(Note: You can change the address at the time of your reply. [Learn more](#))'.

Go to the [Accounts and Import](#) tab of your settings and click on "Add another email address" under "Send mail as".



The screenshot shows the 'Add another email address you own' form. It has a title 'Enter information about your other email address.' with a subtitle '(your name and email address will be shown on mail you send)'. The form contains two input fields: 'Name:' with the value 'Your name' and 'Email address:' with the value 'your-name+tag@gmail.com'. There is a checked checkbox for 'Treat as an alias.' with a link to 'Learn more'. Below this is a link 'Specify a different "reply-to" address (optional)'. At the bottom are two buttons: 'Cancel' and 'Next Step »'.

Afterwards, enter the preferred [display name](#) and subaddress in the new window. You can leave the box "Treat as an alias" checked. (In either case, Gmail asks the recipient to reply to your subaddress, while the main address is used in the [Return-Path header field](#).) Click on the button "Next Step" and you're done. You can now select a different From address the next time you compose a message.

Subaddressing can be useful to filter incoming emails based on their context. Instead of creating several accounts, you can separate different areas of your life with the convenience of having just a single account. Subaddressing also allows you to track whether a company passed your email address on. When you no longer want to receive emails from a company and its affiliates, you can simply block all emails sent to

the address variant you gave them. While subaddressing can be used for creating disposable email addresses on the fly, this protection against abuse can easily be circumvented. If the subaddressing scheme is publicly known, spammers can just remove the tag from customized addresses. A better method against unsolicited messages is to create proper email aliases or forwarding addresses, which are indistinguishable from ordinary addresses. The disadvantage of this approach is that you have to set them up before you can use them. If you use a custom domain for your emails, you might be able to use a so-called catch-all address or customize the subaddressing scheme by using wildcards.

▼ Alias address

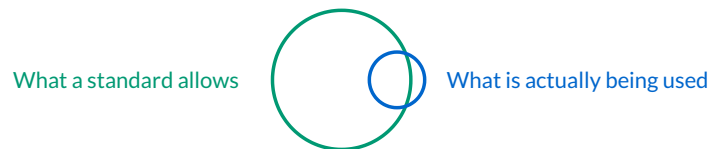
An alias address doesn't have a mailbox associated with it but simply forwards all incoming messages to one or several addresses. The forwarding is done by the incoming mail server of the alias address and the expanded addresses may belong to the same or to different hosts. Unlike in the case of a mailing list, an automatic response by a recipient is sent to the original sender. Alias addresses can forward messages to other alias addresses, which can cause mail loops.

▼ Mailing list

A mailing list is an address which forwards incoming messages to all the subscribers of the list. The administrator of the list can decide who is allowed to send messages to the list and whether each message needs to be approved by a moderator before it is forwarded. Unlike in the case of an alias address, the mailing list software has to change the envelope of the message so that automatic responses from subscribers of the list are sent to the administrator of the list rather than the original sender.

▼ Address syntax

When is an email address valid? As with many technical standards, the answer to this question looks straightforward at first. But as soon as you dig a bit deeper, the answer becomes complicated and messy. What standards allow is often much more than what is widely accepted and used:



Often only a subset of a standard finds adoption, while some things become convention without a formal standard.

The syntax of email addresses is specified in section 3.4.1 of RFC 5322. As mentioned earlier, an address consists of a local part followed by the @ symbol and a domain name. If we restrict ourselves to what is widely adopted, the local part has to consist of the characters a to z, A to Z, 0 to 9, and any of !#\$%&'*+,-/=/?^_`{|}~. A dot . can be used as long as it is between two of the aforementioned characters. In other words, you cannot have multiple dots in a row or at the beginning or end of the local part. The local part has to consist of at least one character, and every mail system must be able to handle addresses whose local part is up to 64 characters long, including any dots. While this is the easy part of the standard, you should avoid most of the special characters if you want to be confident that online services accept your email address. Twitter, for example, accepts only !+_- beyond the alphanumeric characters and the dot. This allows me to sign up with an address such as !+_-@ef1p.com. Gmail, on the other hand, accepts !#\$%&'*+,-/=/?^_`{|}~@ef1p.com as a recipient but fails to recognize this character sequence as an email address in text.

This paragraph is about the complicated part of the standard, which is not widely supported and therefore more of theoretical than practical interest. The local part of an email address can also be a quoted string. Any printable ASCII character is allowed inside of double quotes. If we ignore the obsolete syntax, which may no longer be generated but must still be accepted, the quoted string has to be the whole local part, i.e. it cannot be combined with non-quoted characters. Both "@@ef1p.com" and ".@ef1p.com" are valid addresses, and so is ""@ef1p.com (at least for now). Only " and \ need to be escaped with a backslash in front of them. This means that "\"@ef1p.com and "\"@ef1p.com are also valid addresses. When it comes to whitespace characters, such as space and tab, the situation is a bit confusing. A quoted string can contain escaped spaces (" \" ") through the quoted-pair rule. The only other way a space can be added to a quoted string is as folding whitespace. The standard says that runs of folding whitespace which occur between lexical tokens in a structured header field are semantically interpreted as a single space character. My understanding of this is that a local part with several unescaped spaces (" ") is the same as a local part with a single space (" "). It's not clear to me, though, whether " " is to be interpreted as "". The reason why I think this might be the case is because spaces are clearly excluded from the set of characters which don't need to be escaped. The qtext rule doesn't include the space character, which is %d32 in ASCII, but this might change in the future. If unescaped spaces were meant to have meaning beyond just folding lines, which we'll discuss later, they could easily have been added to the qtext rule. On the other hand, the equivalent

qtextSMTP rule of [RFC 5321](#) does allow spaces. What the standard does clarify is that the escape character \ is semantically invisible. Therefore, "a" and "\a" are equivalent. I assume this means that mail systems are allowed to remove the backslash in front of characters which don't need to be escaped in non-local addresses.

What about the domain part of an email address? While the [Domain Name System](#) allows the use of pretty much any character, the [preferred name syntax](#) requires that each [label](#) consists only of letters, digits, and hyphens, where labels may neither start nor end with a hyphen. SMTP restricts domain names to this syntax. All labels (except the one for the root zone) have to contain at least one character and at most [63 characters](#). The length of the whole domain name is limited to [255 characters](#), including the dots. Domain names are explicitly [case-insensitive](#). Only fully-qualified domain names may be used in email addresses on the public Internet and the domain part of an email address is always written without the trailing dot. The domain name in an email address must have an MX, A, or AAAA resource record. According to [RFC 5321](#), a CNAME record is also permitted as long as its target can be resolved to an IP address through one of the just mentioned record types.

Can an email address use an IP address instead of a domain name? Yes: The [address format](#) allows an IP address in brackets in place of a domain name. For example, user@[192.0.2.123] is a valid email address. However, the SMTP specification says that a host [should not](#) be identified by its IP address, [unless](#) the host is not known to the Domain Name System. One reason for this is that a single mail server can receive emails for multiple domains and the same user might exist in several of these domains. If the recipient address doesn't include a domain name, the mail server might not know to which mailbox it should deliver the message. The domain part of an email address thus serves a similar purpose as the Host header field in [HTTP](#). One might think that mail servers would reject messages with an IP address in the [sender address](#) as [spam](#), but a reader of this article convinced me that this works just fine in many cases. [Apple Mail](#), [Thunderbird](#), and [Gmail](#) also accept such addresses as [recipients](#), while [Outlook.com](#) and [Yahoo! Mail](#) don't.

What about characters outside of the English alphabet? There was a [working group](#) dedicated to the [internationalization of email addresses](#). [RFC 6531](#) defines an [SMTP extension](#) which allows [envelope fields](#) to be encoded in [UTF-8](#) if both the sender and the recipient support it. I'll cover this [later](#).

If you have to validate email addresses, you can use the following [regular expression](#) from the [Living HTML Standard](#): `/^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\. [a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)?*$`. This regular expression allows adjacent dots in the local part but does not allow the local part to be quoted. You could limit the length of the local part with `[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~]{1,64}`, but you should be [liberal in what you accept from others](#). And since [some top-level domains](#) accept email, the regular expression intentionally ends with `*$` instead of `+$`. As we will see later on, the [validation of internationalized domain names](#) is much more difficult.

▼ Common addresses

If you use your own domain for email, you can choose the [local part](#) of your addresses however you want as long as you adhere to the [address syntax](#). Some local parts, though, are commonly used to reach the person with a specific role in an organization:

Address	Expectation
info@	Reach someone from the administrative office.
contact@	Be directed to the desired person within the organization.
sales@	Receive purchase information from the sales person.
support@	Get support for the offered product or service.
marketing@	Provide feedback to marketing campaigns.
abuse@	Report inappropriate public behavior.
security@	Responsibly disclose a security vulnerability .
postmaster@	Reach the email administrator (required according to RFC 5321).
hostmaster@	Reach the DNS administrator.
webmaster@	Reach the Web administrator.
admin@	Reach the technical administrator (as an alternative to the previous three addresses).

Most of these addresses are encouraged by [RFC 2142](#): "Mailbox names for common services, roles, and functions". Role-based addresses are usually configured as [aliases](#) so that incoming emails can be forwarded to several people.

Recipients

You can address the recipients of a message in [three different ways](#):

- The To field contains the address(es) of the primary recipient(s). As a sender, you expect the primary recipient(s) to read and often to react to your message. The expected reaction can be a reply or that they perform the requested task.
- The Cc field contains the address(es) of the secondary recipient(s). As a sender, you want to keep the secondary recipient(s) informed without expecting them to read or react to your message. (Cc stands for carbon copy.)
- The Bcc field contains the address(es) of the hidden recipient(s). Their address(es) are not to be revealed to other recipients of the message. The field is usually fully preserved in your folder of sent messages but fully removed in the version of the email that is delivered to others. Alternatively, a different message could be delivered to each hidden recipient where their address alone is listed in the Bcc field. The standard also allows hidden recipients to see each other; they just have to be removed for the primary and secondary recipients. The vague semantics of this feature leads to several problems. (Bcc stands for blind carbon copy.)

Important: Just because someone is listed as another recipient doesn't mean that they received the same message as you. The reason for this could be innocuous or malicious. On the one hand, it may be that the email could simply not be delivered to them. On the other hand, the sender might have delivered the message only to you in order to mislead you. Your email service provider has no way of verifying that the same message has also been delivered to the other recipients. This allows a fraudster to fake a relationship that they do not have or to lead you to believe that they have done the introduction you asked them for, even when this is not the case. If you reply to all, your reply would also be sent to the faked recipients, of course.

▼ Group construct

The address specification allows senders to group addresses with the following syntax: {GroupName}: {ListOfAddresses};, where the curly brackets have to be replaced with actual values. ListOfAddresses is a comma-separated list of addresses, where each address can also have a display name. You can send an email to several groups, but you cannot nest groups. The list of addresses can be empty, which allows the sender to hide the recipients of a message. Even though the To field is optional and can therefore be skipped completely, some mail clients prefer to put something like `undisclosed-recipients:;` into this field when you list all the recipients in the Bcc field. As far as I can tell, this is the primary use of the group construct nowadays.

Sender

There are two relevant fields to indicate the originator of a message:

- The From field contains the address of the person who is responsible for the content of the message.
- The Reply-To field indicates the address(es) to which replies should be sent. If absent, replies are sent to the From address.

Important: The core email protocols do not authenticate the sender of an email. It's called spoofing when the sender uses a From address which doesn't belong to them. Forged sender addresses are a huge problem for the security of email. There are additional standards to authenticate emails. For them to have the desired effect, though, both the sender and the recipients have to use them.

▼ Sender field

RFC 5322 differentiates between the author and the sender of a message. Usually, the person who writes the message is also the one who sends it. If the author and the sender are different, the sender should be provided in the Sender field. The standard also allows several addresses in the From field. If this is the case, the email has to include a Sender field with a single address. However, I'm not aware of any mail clients which support this. In practice, the addresses of the co-authors are simply added to the Cc field. Their contribution is made clear to the primary recipients by mentioning the names of all the authors at the end of the message. Remember that a sender can lie about their co-authors: Just because the address of a person is included in the Cc field doesn't mean that the email has been delivered to them and that they agree with the content of the message.

▼ No reply

Many emails are sent from automated systems, which cannot handle replies. Examples of such emails are notifications about events on a platform and reports about some usage statistics. RFC 5322 demanded that there is a From field with one or several addresses. RFC 6854 updated the standard in 2013 to allow the group construct to be used in the From field as well. This allows automated systems to provide no reply address by using an empty group in the From field, rather than having to rely on users interpreting an address such as `no-reply@example.com` correctly. The automated system can still identify itself by choosing the name of the group appropriately, for example `LinkedIn Notification Bot:;`. In the absence of an alternative to indicate the originating domain to the user, I strongly advise against using an empty group in the From field, though, because this defeats all efforts towards domain authentication. Even the RFC itself recommends against the general use of this method and says that it is for limited use only. Thus, we still have to wait for a usable No-Reply header field, unfortunately. (The empty group construct is used to downgrade internationalized email addresses as specified in RFC 6857.)

Subject

The Subject field identifies the topic of a message. Its content is restricted to a single line but the line can be of arbitrary length. (We'll talk about [encoding](#) later.) [RFC 5322](#) also defines other informational fields, namely Comments and Keywords, but I've never seen them being used. All informational fields are optional, which means an email doesn't need a subject line. The mail clients I've checked, though, include the Subject field even when it's empty. While the message is transmitted with an empty Subject field, mail clients usually display "(No subject)" instead of nothing.

▼ Prefixes

When you reply to a message, your mail client automatically suggests the new subject: "Re: " followed by the original subject. While I would argue that "Re" stands for "reply", [RFC 5322](#) says that it is an abbreviation of the Latin "in re", which means "in the matter of". Similarly, if you forward an email to another recipient, your mail client typically puts "Fwd: " in front of the original subject. Using such prefixes in replies and forwarded emails is optional. In particular, they have no technical significance. As we will see [later](#), messages are grouped into conversations based on other, more reliable information.

Body

Last but not least, an email has a [body](#) (which is strictly speaking optional). The body contains the actual content of a message. It can be formatted in different ways and can consist of [different parts](#). Splitting the body into several parts is useful, for example, to send a plaintext version alongside an [HTML-encoded message](#) or to [attach files](#) to an email. We'll discuss [later](#) how all of this works.

▼ Size limit

The email standards impose [no size limit](#) on messages. Since various servers have to store your message at least temporarily, they are configured to reject messages larger than a certain size. Many providers have a size limit between around [25 to 50 MB](#). Even if your email service provider allows you to send larger messages, such messages might still be rejected by the mail server of the recipient. Since attachments have to be encoded in a [particular way](#), their original size can be at most around 70% of the actual size limit.

Architecture

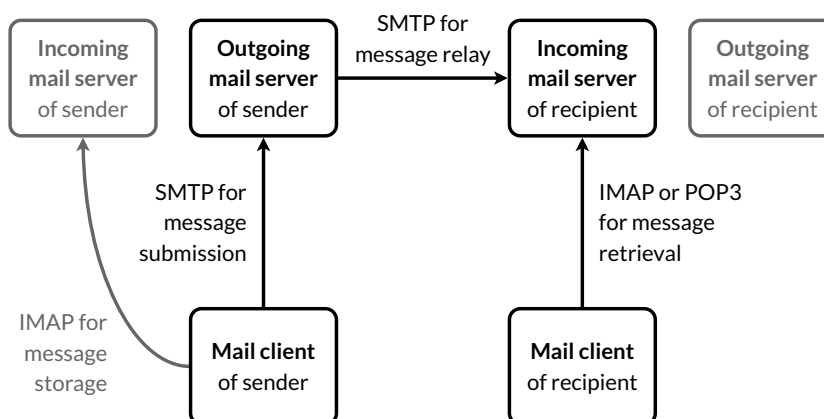
There are four separate aspects to understand email from a technical perspective:

- **Format:** What is the syntax of email messages?
- **Protocols:** How are these messages transmitted?
- **Entities:** Who transmits these messages to whom?
- **Architecture:** How are these entities arranged?

Let's go through them one by one in the opposite order.

Simplified architecture

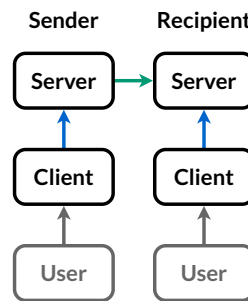
One reason why email is so hard to grasp is because the [official terminology](#) is unnecessarily complicated in most circumstances. Throughout this article, we'll work with a much simpler version. Email follows the [client-server model](#): A client opens a [connection](#) to a server in order to request some service. In all the graphics where arrows represent an exchange of data, the arrows point from the client to the server; i.e. in the direction of the request, not the response. The following entities and protocols are involved in the transmission of a message from a sender to a recipient:



The simplified email architecture. We'll discuss each entity in the [next section](#) and the protocols [thereafter](#).

▼ Standardization

If we ignore for a moment that there are separate servers for incoming and for outgoing mail, we're left with the following: The user interacts with a client to read and compose messages. The client submits the composed messages to a server for delivery. The client also fetches newly received messages from the server. The server connects to other servers in order to deliver some messages. The important thing to note is that the interactions between these entities are independent from one another:



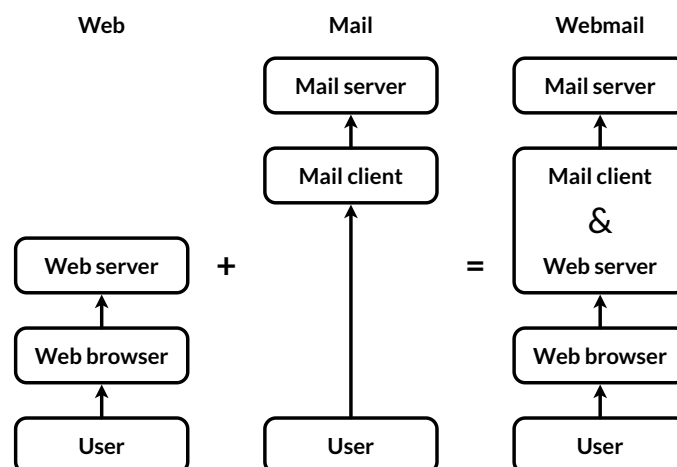
How emails are submitted and accessed (in blue) is independent from how emails are exchanged between servers (in green).

Let's have a look at each of these interactions with regard to standardization:

- **Server → server:** Just as any machines on the Internet can communicate with one another (as long as we ignore firewalls), any users with an email address can send each other messages (as long as we ignore spam filters). This works only because the exchange of messages between mail servers is standardized. Anyone who adheres to this standard can participate in the global email system. In order to maintain compatibility with older servers, support for new functionality is always optional.
- **Client → server:** How clients submit and access emails doesn't have to be standardized for email to remain interoperable according to the previous point. Luckily, we do have open standards for accessing one's mailbox. Since these standards are older than all commercial email service providers, most email service providers support at least one of them. This has the advantage that you can switch the server without switching the client and that you can switch the client without switching the server. This reduces vendor lock-in on both the client- and the server-side, which leads to more choice for consumers. However, email service providers can still support proprietary features, which only their client knows how to make use of.
- **User → client:** How users interact with mail clients is not standardized. In particular, users don't have to sit directly in front of their mail client. They can also interact with a mail client over the Web, for example. Some standards demand that certain actions have to be confirmed or initiated by the user. Apart from this, mail clients are free to present information to the user in any way they want. But similar to how you can drive a car from any brand if you know how to drive a car from one brand, users have developed expectations regarding how the above concepts are presented. For example, Cc is always called Cc.

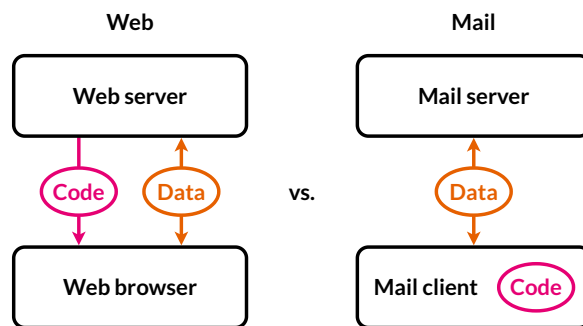
▼ Webmail

Email service providers usually offer a web interface to their email service. Instead of configuring a mail client, which runs locally on your device, you can visit a provider-specific website with a web browser in order to access your messages. This is known as webmail and it has the advantage that you can read and compose emails from any device with a web browser. From the perspective of email standards, this constitutes a remote access to the mail client:



In the case of webmail, the mail client is accessed via a web server using a web browser.

Unlike a dedicated mail client, which typically stores the downloaded messages in the persistent memory of your device for offline access, you have to be connected to the Internet to use webmail. While not generally desirable, fetching all data only temporarily until you log out is useful when you want to access your emails from someone else's device. In addition, configuring a mail client is more complicated than navigating to a website. This might explain the popularity of webmail. In my opinion, the biggest disadvantage of webmail is that the logic of how you can interact with your messages comes from the provider:



On the left, the code to interact with the data comes from the server.
On the right, the logic is inside the client and only data is exchanged.

If you need additional features, such as end-to-end encryption or interaction with a service from a different provider, you have to find workarounds with browser extensions. Open-source mail clients, on the other hand, can be modified at will. In order to give you more control over your messages, most email service providers offer an application programming interface (API) for access to your mailbox, such as IMAP or POP3. In the case of Gmail, you have to enable the API through the web interface for your account before a mail client can use it.

When it comes to security, there's no clear winner. Webmail has the advantage that you always run the newest version of the code, which is sandboxed from the rest of your system by the web browser. On the downside, attacks like phishing, cross-site scripting, and cross-site request forgery are only possible because the browser runs untrusted code, which a dedicated mail client doesn't. Whether you access your emails via the Web or via a local mail client is a matter of individual preference.

As we've learned in the previous box, how users interact with their mail client isn't standardized. Webmail is thus of no interest for the scope of this article. All you need to know is that email has nothing to do with the Web. Both are independent services that run over the Internet. Moreover, email is older than the Web: SMTP was first defined in 1982, POP in 1984, and IMAP in 1986. The HyperText Transfer Protocol (HTTP), which underpins the Web, was introduced around 1990.

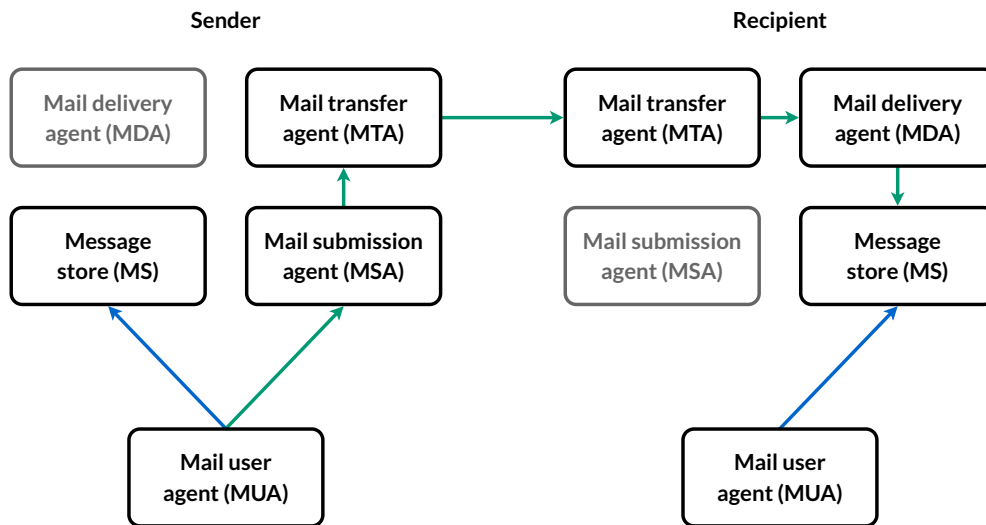
Official architecture

For the sake of completeness and to enable you to understand the linked articles, this subsection covers the official terminology as used, for example, in RFC 5598. In the official documents, there are five instead of three entities, with each of them having a more complicated name and, of course, an associated three-letter acronym (TLA):

TLA	Name	Description
MUA	<u>Mail user agent</u>	Client to compose, send, receive, and read emails, such as <u>Microsoft Outlook</u> , <u>Apple Mail</u> , and <u>Mozilla Thunderbird</u> .
MSA	<u>Mail submission agent</u>	Server to receive outgoing emails from authenticated users and to queue them for delivery by the mail transfer agent (MTA).
MSS	<u>Mail submission server</u>	
MTA	<u>Mail transfer agent</u>	Server to deliver the queued emails and to receive them on the other end. It then forwards the received emails to the mail delivery agent (MDA).
MDA	<u>Mail delivery agent</u>	Server to receive emails from the local mail transfer agent (MTA) and to store them in the message store (MS) of the recipient.
MS	<u>Message store</u>	Server to store the emails received from the mail delivery agent (MDA) and to deliver them to the mail user agent (MUA) of the recipient.
MAS	<u>Mail access server</u>	

The terminology used by the Internet Engineering Task Force (IETF) in its official documents, such as this one. The terms in *italics* are used in some newer documents, such as this one. I added them because I like them better.

These terms are not as precise as they seem to be and the boundaries are often fluid in practice. Having more entities also changes the architecture. What follows is a nicer version of this ASCII graphic, which is a masterpiece to be appreciated in its own right.



The official [Internet Mail Architecture](#) with SMTP connections in green and IMAP connections in blue.

None of the servers have to be a single machine. In addition, the incoming MTA and the outgoing MTA don't have to be the same.

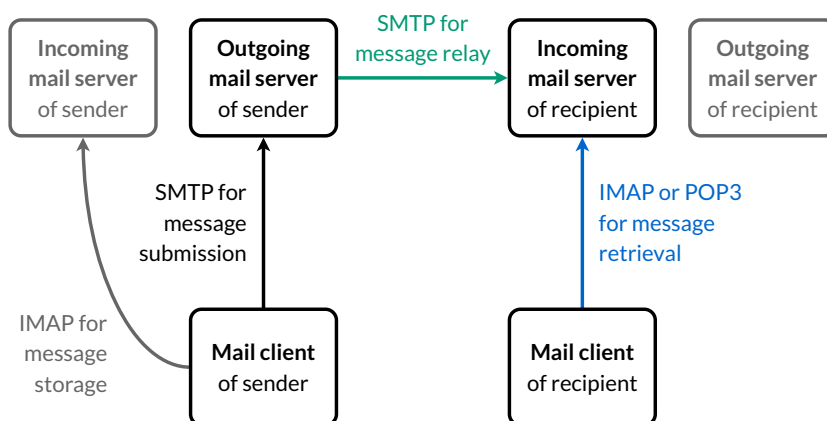
Entities

There are three entities in the simplified architecture: the mail client, the outgoing mail server, and the incoming mail server.

Mail client

The mail client is a computer program to compose, send, retrieve, and read emails. It provides the interface through which users handle email. The mail client runs either locally on the user's device or remotely on a web server. Examples of the former kind are [Microsoft Outlook](#), [Apple Mail](#), and [Mozilla Thunderbird](#). Examples of the latter are [Google Gmail](#) and [Yahoo! Mail](#) when accessed through a web browser. (Both companies also provide [mobile apps](#) for [Android](#) and [iOS](#), which fall into the former category.)

The mail client connects to the outgoing mail server to submit messages for delivery to other users and to the incoming mail server to fetch new messages from the user's mailbox. Both servers authenticate the user, typically with a username and a password. The mail client connects to the incoming mail server through a different interface than outgoing mail servers do, which can be seen on the recipient's side of the [simplified mail architecture](#):



The recipient's mail client connects to the incoming mail server using a different port and protocol than outgoing mail servers. It's usually also a different machine with a different domain name and IP address than the one outgoing mail servers connect to.

This distinction is apparent in the [official mail architecture](#), where the message store (MS) and the mail transfer agent (MTA) reside in different boxes. By giving the impression that the incoming mail server is a single machine, the simplified model doesn't explain why the incoming mail server needs to be configured in the mail client of its user but not in the outgoing mail servers of other users. Since the simplified architecture is less confusing in every other regard, it's still the preferred model for the scope of this article.

▼ Configuration

When you add an email account to your mail client, you usually have to configure the incoming mail server and the outgoing mail server manually, unless you use a [popular email service provider](#). If manual configuration is required, you have to look through the documentation of your email service provider to find the domain names of the two servers and then copy the information to the respective fields in your mail client. While most email service providers use the [default port numbers](#), which means that you typically don't have to configure them, the domain names of the two servers aren't standardized. It's often the case that their addresses are subdomains of the domain after the @ symbol in your email address, such as `imap.gmail.com` and `smtp.gmail.com` for `@gmail.com`. However, many organizations don't host their emails themselves, in which case the domains of the two servers are likely completely different from the organization's domain. This is the case for my email configuration:

The screenshot shows the 'Server Settings' tab for an email account named 'EF1P'. It is divided into two sections: 'Incoming Mail Server (IMAP)' and 'Outgoing Mail Server (SMTP)'. Both sections have fields for 'User Name', 'Password', and 'Host Name'. The 'User Name' for both is 'kaspar@ef1p.com'. The 'Host Name' for both is 'mail.gandi.net'. There are checkboxes for 'Automatically manage connection settings' which are checked. There is also a button for 'Advanced IMAP Settings'.

Section	Field	Value
Incoming Mail Server (IMAP)	User Name	kaspar@ef1p.com
	Password
	Host Name	mail.gandi.net
Outgoing Mail Server (SMTP)	User Name	kaspar@ef1p.com
	Password
	Host Name	mail.gandi.net

The simplified email architecture corresponds to what mail clients like Apple Mail display to you. The domain of the address (`ef1p.com`) is different from the domain of the servers (`mail.gandi.net`). The host names of the incoming mail server and the outgoing mail server are usually not the same.

One more thing that users need to be informed about is whether to use the full email address or only the local part before the @ symbol (or even something completely different) as the username. While flexibility is great for customizing a setup to the particular needs of an organization, it also leads to an unnecessarily complicated experience for users.

▼ Custom domains

Please note that you cannot simply set up [CNAME records](#) in your own domain for the incoming and outgoing mail servers if you want to avoid instructing your users to use an external domain because the [TLS certificates](#) used by the email service provider would no longer match. For example, if I point `imap.ef1p.com` to `mail.gandi.net` with a CNAME record in my DNS zone and use the former in the [server settings](#), then my mail client expects the TLS certificate to be issued for `imap.ef1p.com` and aborts the connection when it receives a certificate for `mail.gandi.net`. Besides vanity, such a setup could be desirable because it would allow the IT administrator of an organization to migrate all messages to another email service provider without requiring every member of the organization to change their email settings. This can be realized with the technique that I cover in the [next box](#).

▼ Autoconfiguration

Wouldn't it be nice if mail clients could configure themselves automatically by fetching the required information directly from the email service providers? The good news is that we have a standard for exactly this purpose. The bad news is that almost no one is using it, even though it's simple and elegant. [RFC 6186](#) defines how to use [SRV records](#) in the [Domain Name System \(DNS\)](#) for locating email submission and access services. Using DNS for fetching the required information is elegant because the [email protocols](#) already depend on DNS, the information is provided by the owner of the domain, and the system scales well thanks to the caching of answers by intermediary resolvers.

However, if the answers are not authenticated with [DNSSEC](#), an attacker who can spoof DNS responses can direct the mail client to malicious servers. This attack vector is really bad because TLS doesn't prevent it (the malicious servers can have valid certificates for their domains) and because ~~passwords are often transmitted in cleartext~~ over the encrypted channel instead of using non-reusable [challenge-response authentication](#), such as [SCRAM](#). The attacker can thus authenticate as the user to the legitimate servers beyond the duration of the attack until the user changes their password. The [RFC just says](#) that the domain names of the servers should be confirmed by the user if they are not subdomains of the queried domain without requiring or even mentioning DNSSEC. As everyone working in IT security knows, security-critical decisions should not be left to users.

[RFC 2782](#) specifies the format of service (SRV) records. The basic idea is to use a different subdomain for each protocol and service and list the port number and domain name of the host which provides the particular service in the data field of the resource record. The subdomain is constructed as follows: `_service._protocol.domain`, where `domain` is the domain part of the email address, `_protocol` is `_tcp`, and `_service` is `_submission/_submissions`, `_imap/_imaps`, or `_pop3/_pop3s` in the case of email. The data of SRV records consist of a priority number, a weight number, a port number, and the domain name of the target host separated by a single space. If several records are returned, the client has to connect to the host with the lowest priority number first and fall back to the host with the next higher priority number only if all hosts with lower priority numbers are unreachable. If there are several records with the same priority, the client should select one at random proportionally to its weight. This can be useful to [balance the load](#) among several hosts. If there isn't any server selection to do, then the weight should be set to zero. For example, if you have the [dig command](#) installed in your [command-line interface \(CLI\)](#), executing `dig srv _submission._tcp.gmail.com +short` returns `5 0 587 smtp.gmail.com.`. This means that mail clients should submit outgoing emails to `smtp.gmail.com` on port 587 when using `gmail.com`. If the host name is `.`, the service is explicitly not available at this domain. For example, running `dig srv _imap._tcp.gmail.com +short` returns `0 0 0 .` because Gmail supports IMAP only with [Implicit TLS](#), which is usually called IMAPS.

You can check the email service records of a domain with the following tool, which uses an [API by Google](#) for its DNS queries:

Domain:   

If you played around with the above tool for a while, you might have realized that not many domains have service records for email. Probably for this reason, none of the major mail clients actually [use this autoconfiguration method](#). In my opinion, this is really unfortunate but not surprising given that only supply can generate demand and only demand can generate supply.

I can see only three potential weaknesses with this standard:

1. Service records make the incoming and outgoing mail servers publicly known. For public mail services, where anyone can create an account, this is the case anyway. For private mail services, on the other hand, such knowledge makes attacks on the infrastructure easier if the mail servers cannot be guessed otherwise.
2. Service records provide no information about the [username](#) and the [authentication method](#). The latter can be discovered, though, simply by connecting to the server and inquiring about its [supported extensions](#).
3. The provided information cannot depend on the local part of the email address since DNS queries don't support additional parameters. There are autoconfiguration protocols which support this, such as [the one used by Thunderbird](#).

Besides improving the experience of users, service records make it possible to migrate an organization to another email service provider without requiring every member of the organization to change their email settings. According to [RFC 6186](#), mail clients should cache the resolved hosts until they can no longer establish a connection or user authentication fails. When either of these happen, mail clients are supposed to fetch the SRV records of the same `_service` again. Mail clients may not switch from IMAP to POP3 or vice versa without the user's consent.

If you want to configure SRV records for your domain, you can put the following entries into your [zone file](#):

```
_imap._tcp 10800 IN SRV 0 0 143 {Domain}.
_imaps._tcp 10800 IN SRV 0 0 993 {Domain}.
_jmap._tcp 10800 IN SRV 0 0 443 {Domain}.
_pop3._tcp 10800 IN SRV 0 0 110 {Domain}.
_pop3s._tcp 10800 IN SRV 0 0 995 {Domain}.
_sieve._tcp 10800 IN SRV 0 0 4190 {Domain}.
_submission._tcp 10800 IN SRV 0 0 587 {Domain}.
_submissions._tcp 10800 IN SRV 0 0 465 {Domain}.
```

Insert the appropriate Domain and use `0 0 0 .` for all the services which are not supported by your email service provider.

▼ Configuration database

At this point, you may be wondering how mail clients can often figure out the correct configuration by themselves despite the lack of an established standard. Most mail clients look up the configuration for popular email service providers in a database, which is either delivered with the client or centrally hosted by the software manufacturer. Some mail clients also use custom autoconfiguration protocols, which typically fetch an XML file hosted at a specific subdomain via HTTPS.

Let's have a look at how Thunderbird does it. It's autoconfiguration process is well documented and it's configuration database is free to use for any mail client. Given an email address {Address} = {LocalPart}@{Domain}, Thunderbird goes through the following steps from top to bottom until it finds a suitable configuration:

1. Check the installation directory for a configuration file. This is useful for when the employer administrates the user's device.
2. Check `https://autoconfig.{Domain}/mail/config-v1.1.xml?emailaddress={Address}` for a configuration file. Unlike the mechanism discussed in the previous box, this file can be generated dynamically based on the email address. This is useful for when the username is neither the email address nor the local part.
3. Check `https://{Domain}/.well-known/autoconfig/mail/config-v1.1.xml`. The key difference between this and the previous lookup is that the autoconfig subdomain in step 2 can point to a web server operated by your email service provider, while the lookup in the current step must be handled by the Domain itself.
4. Look for a configuration file in the central database at `https://autoconfig.thunderbird.net/v1.1/{Domain}`.
5. Look up the MX record of the domain in the Domain Name System and then check whether the central database has an entry for the so-called apex domain at the root of the zone. This is useful for custom domains like `ef1p.com`, which has an MX record pointing to `spool.mail.gandi.net`, which belongs to the zone starting at `gandi.net`. The central database has an entry for gandi.net, which is how Thunderbird would find the configuration for my email address.
6. If all previous attempts to find a configuration failed, Thunderbird resorts to guessing the mail servers. It tries to connect to common server names such as `mail.{Domain}`, `smtp.{Domain}`, and `imap.{Domain}` on the default port numbers and checks whether they support TLS or STARTTLS and the challenge-response authentication mechanism (CRAM). The last check prevents Thunderbird from accidentally revealing the user's password to the wrong server. Unfortunately, CRAM is rather weak. The far better salted challenge-response authentication mechanism (SCRAM) should be used instead.
7. If all of the above steps fail, the user has to enter the configuration themselves.

I've implemented steps 2 to 5 of Thunderbird's discovery procedure in case you need to configure a mail client and don't know the required information. The tool makes requests to the entered domain according to the above description and, if necessary, to Thunderbird's database. If the fifth step is also needed, the DNS queries are made with Google's DNS API. Please note that the requests are sent directly from your browser, which means that the lookups fail if the server does not allow cross-origin resource sharing (CORS) with an Access-Control-Allow-Origin header field value of *. Since such a header field is not required for mail clients, this is often not the case. For this reason, the protocol tools query only Thunderbird's database.

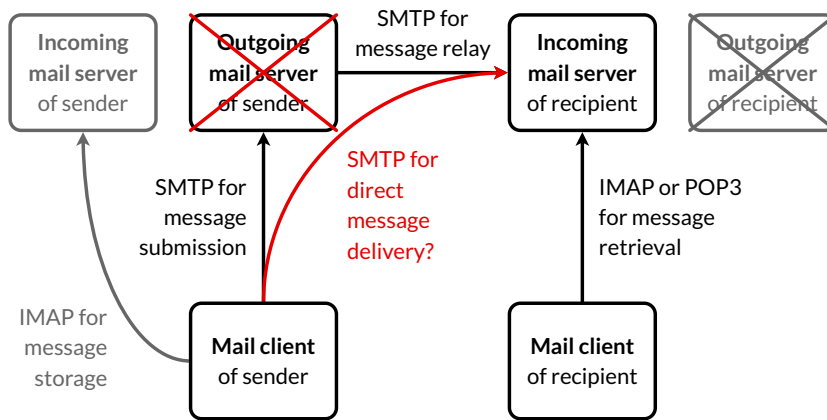
Domain:   

Outgoing mail server

The outgoing mail server accepts messages from mail clients and queues them for delivery. It then determines the incoming mail server of each recipient and delivers the message to them. The outgoing mail server acts as a server in the interaction with mail clients but assumes the role of a client when relaying the message to incoming mail servers. (Connections are always initiated by clients.) If the outgoing mail server cannot deliver a message, it sends a bounce message to the user who submitted the message. While the outgoing mail server should not change the content of a message, it adds information about the submitter at the top. Before accepting a message, the outgoing mail server authenticates the user, typically based on a username and a password.

▼ Why do we need outgoing mail servers when mail clients could simply deliver the messages directly?

Before we discuss why we need outgoing mail servers, let's first have a look at what the modified architecture would look like:



A hypothetical email architecture without outgoing mail servers.

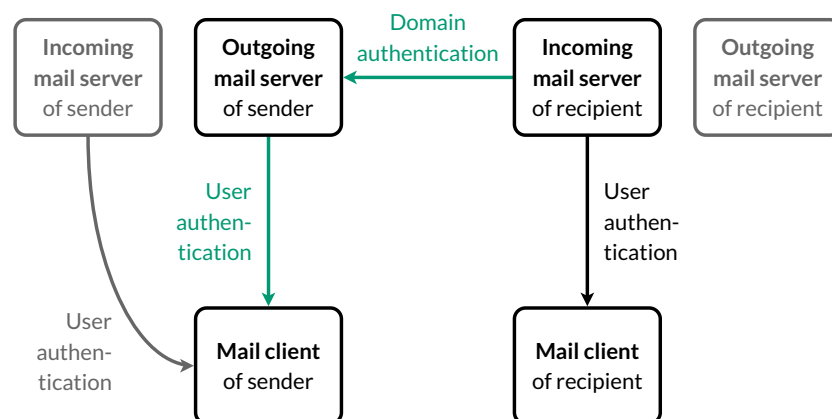
Since outgoing mail servers are just a piece of software and can thus be integrated into mail clients, it is technically possible to send emails directly to the incoming mail server of each recipient. In fact, sending an email to someone ~~from the command line~~ is my favorite demonstration in the seminars I give. Only badly configured incoming mail servers accept such messages, though.

There are two main reasons why outgoing mail servers are used in practice:

- **Shift work from the client to the server:** Unlike the mail client, which runs on the user's device, the outgoing mail server typically has a fast and permanent Internet connection. For example, when you send an email from your smartphone, your Internet connection might be slow and also expensive due to roaming. When you switch off your smartphone on an airplane or overnight, its mail client is offline for several hours. Thus, it makes sense to implement the following features on a server:
 - **Retry after unsuccessful delivery:** As we will see in the next subsection, an incoming mail server can reject a message for a number of reasons. One reason is simply to deter spammers, who often won't attempt to transmit the message again. An incoming mail server might also be unreachable due to maintenance or malfunctioning. While Internet outages are rare in most areas of the world, it might happen that a communication link is temporarily unavailable. This is why the standard demands that messages which cannot be delivered immediately have to be queued and their transmission retried by the sender after a delay of at least 30 minutes for several days.
 - **Send a single message to several recipients:** If you send an email to several recipients, your mail client submits the email only once to the outgoing mail server. The outgoing mail server then delivers a copy of the email to each recipient. This is especially useful when you send a big attachment to many recipients over a bad Internet connection.
 - **Batch messages for delivery:** In the early days of email, access to the Internet was expensive and you often paid for the duration of your connection rather than for the volume of transmitted data. Since machines were permanently connected only in the local network of your organization, it made sense to collect outgoing mail from members on a local server and then deliver the messages once a link had been established. Given that most organizations pay a flat rate for their Internet access nowadays, this aspect is only of historic relevance.
- **Reduce spam and phishing:** Unsolicited mail is an annoyance, both in the analog and the digital world. Unless we impose a cost on the sender, it's impossible to eliminate spam completely in a decentralized system in which everyone is allowed to participate. Being able to spoof the sender of an email, which is often used for phishing, is a real security concern. System administrators deploy the following measures to curb the two problems, which require the use of outgoing mail servers:
 - **Blocked connections:** Incoming mail servers listen on port 25 for new messages. An Internet service provider (ISP) can prevent emails from being sent from its network by blocking all outgoing connections with a destination port of 25. Its customers can still connect to an outgoing mail server on port 587, which has to be in a different network or explicitly whitelisted by the ISP in order to be able to deliver messages on behalf of its users. This measure makes it technically infeasible to send emails directly to the incoming mail server of a recipient. Many Internet hosting providers also block outgoing traffic on port 25 by default to fight spam and to protect the reputation of their IP address range. For some providers, such as Linode, you can contact their customer service to lift this restriction, for other providers, such as DigitalOcean, the restriction is permanent.
 - **Address reputation:** Incoming mail servers learn the sources of legitimate email over time. Messages coming from such sources are likely to be delivered to the user's inbox. Messages from sources with a bad reputation are often dropped on arrival. Messages from unknown sources are either dropped or put into the user's spam folder. Reputation is crucial to build trust among unverified participants. Even when the sender of an email is authenticated, reputation remains at the core of any effort to fight spam. As we will see later on, you have to buy into the reputation of others if you want to have your emails delivered reliably to your customers. A whole industry has developed around this value proposition. Since building a reputation as a trustworthy email sender yourself is too much of a struggle for most Internet users and companies, the port restriction mentioned in the previous bullet point isn't much of a problem in practice.
 - **User authentication:** Email service providers are incentivized to protect their reputation because users would no longer use their service if emails are no longer delivered reliably. This is why email service providers impose sending limits on their users and delete

accounts when misbehavior is reported to them, which is possible only if they authenticate their users before relaying messages. For example, Gmail limits the number of messages per day to 2'000 and the number of recipients per message to 100 if the message is submitted from a mail client rather than the web interface. Vouching for users could also be done differently, for example by delegating trust to mail clients with digital signatures. However, an email service provider could no longer rate limit and filter outgoing messages if mail clients delivered them directly.

- **Domain authentication:** When it comes to information security, trust is good but control is better. Spam is a problem of quantity: You simply want to bring the volume of unsolicited messages to a bearable level. Phishing, on the other hand, is a problem of quality: A single successful attack can cause a lot of damage. A reputation system is great for fighting spam but not good enough for fighting phishing. The email delivery protocol itself doesn't prevent the sender from putting an arbitrary address into the From field. In the absence of a mechanism to authenticate the sender, you can only hope that email servers with a good reputation don't misuse their reputation and send messages with spoofed sender addresses and malicious content to you. The idea behind domain authentication is that each domain owner can specify which outgoing mail servers are allowed to send messages from their domain. Incoming mail servers can then verify whether the sender of a message is indeed authorized to send messages from the claimed domain. In combination with user authentication, where outgoing mail servers prevent their users from sending messages in the name of another user at the same domain, the two mechanisms guarantee that the sender of a message owns the claimed From address. There would be other ways to achieve a similar result without requiring outgoing mail servers, but this is how email works.

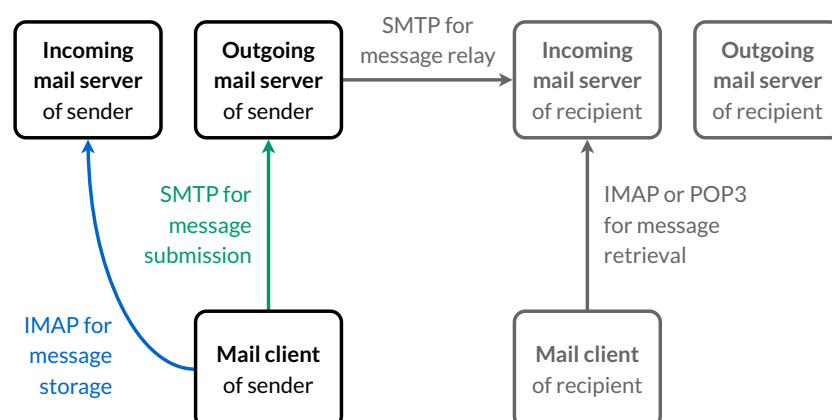


The incoming mail server verifies that the outgoing mail server is authorized to send messages from the claimed domain, while the outgoing mail server of the sender ensures that each user uses their own address in the From field.

As we will see in the next box, having an audit trail of sent emails is not among the reasons why outgoing mail servers are used. And while an outgoing mail server could be useful to hide your IP address from the recipients, many outgoing mail servers leak your IP address in a Received header field. Privacy could be one of the reasons for using an outgoing mail server but often isn't.

▼ How to avoid submitting the same message to both the outgoing mail server and the incoming mail server?

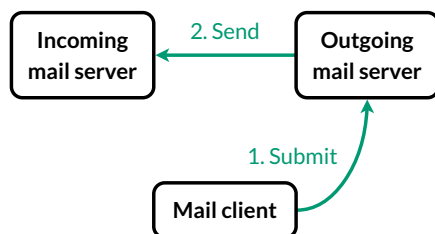
If you want to keep a record of all emails that you've sent, your mail client has to store each outgoing message in the sent folder on your incoming mail server. Since we focussed on how an email gets to its recipient so far, this aspect has been grayed out in the above architecture diagrams. In most cases, the client has to submit the same message twice: Once to the outgoing mail server for delivering the message to the recipients, and once to the incoming mail server for updating the sent folder.



The mail client submits the same message to both the outgoing mail server and the incoming mail server.

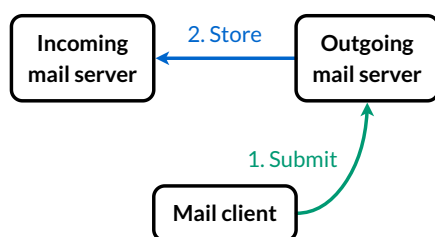
For a bandwidth-limited mail client, this is not ideal. There are four different approaches to avoid this double submission:

- **Always Bcc yourself:** You can configure most mail clients to add yourself as a Bcc recipient whenever you compose an email. The outgoing mail server then delivers a copy of each message to your inbox. The downside of this method is that your copy doesn't include the other Bcc recipients. Moreover, sent and received messages aren't separated, which may be desirable.



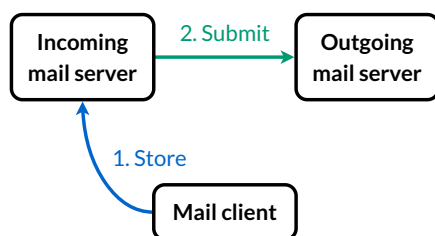
The outgoing mail server sends a copy to your inbox.

- **Gmail:** Google's outgoing mail server automatically stores a copy of sent messages in the user's sent folder. In order not to end up with duplicates in the sent folder, the mail client shouldn't store sent messages in the user's mailbox. Since the mail client cannot detect this non-standard behavior when submitting a message to the outgoing mail server, either the mail client has to treat `@gmail.com` addresses differently or the user has to disable the option to save a copy in the sent folder manually. Since mail clients remove the Bcc field before submission, Gmail recovers it from the envelope of the message.



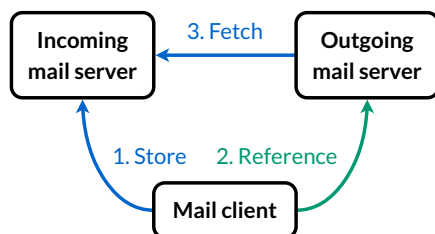
Gmail automatically stores sent messages.

- **Courier-IMAP:** The Courier Mail Server has a configuration option to designate a mailbox folder as a special outbox folder. When the mail client stores a message in this folder, the server sends the message to the addresses listed in the To, Cc and Bcc fields. What makes this approach interesting is that a mail client can use IMAP for everything and no longer needs to support SMTP. Unfortunately, this feature is also not standardized and mail clients can therefore not rely on its availability.



The Courier IMAP server can deliver emails.

- **Lemonade profile:** The only standardized solution to the double submission problem is a collection of extensions to IMAP and SMTP submission, which is called the lemonade profile. The URLAUTH extension to IMAP allows mail clients to create references to mailbox data, which include the required authorization to access the data. The BURL extension to SMTP submission allows mail clients to instruct the outgoing mail server to fetch data from the user's mailbox. If the mail servers support these two extensions, the mail client can upload the message to be sent to the user's mailbox on the incoming mail server and then instruct the outgoing mail server to deliver this message.



The lemonade profile enables the outgoing mail server to fetch content for delivery from the incoming mail server.

The lemonade profile includes additional extensions, such as the [CATENATE extension](#) to IMAP and the [PIPELINING extension](#) to SMTP. The former allows mail clients to compose new messages based on existing messages directly on the IMAP server. This makes it possible to forward large attachments without having to download and upload them first. The latter allows clients to send several commands in a row without having to wait for a response from the server between them. This reduces the number of [round trips](#), which makes communication over large distances much faster.

Incoming mail server

The incoming mail server waits for connections from outgoing mail servers of other users. When an outgoing mail server connects to transmit a message, the incoming mail server records the message together with ~~other information from the session~~, such as the sender's IP address. The incoming mail server can reject the incoming message for a number of reasons: The recipient might not exist, their mailbox might be full, the message might be ~~too long~~, or the sender might not be ~~trusted~~. If the message is rejected, the outgoing mail server can either try to retransmit it at some later point or inform the user about the ~~failed delivery~~. If, on the other hand, the incoming mail server accepts the message, it also assumes responsibility for delivering the message. If it fails to do so, for example when the message needs to be ~~forwarded~~, then the incoming mail server should notify the author of the message.

Once the session with the outgoing mail server is over, the incoming mail server adds the ~~additional information~~ collected during the session to the top of the accepted message. It then evaluates whether the message is likely ~~spam~~. Depending on the score of this evaluation, the message is either delivered to the recipient's inbox, quarantined to the recipient's spam folder, or discarded without notifying the author. While the last option violates the principle that mail is either delivered or returned, the alternative is often ~~worse~~. This is why the standard [explicitly allows](#) incoming mail servers to drop received messages silently. If the receiving address is an ~~alias~~, the incoming mail server forwards the message to the configured email address instead of delivering it to an inbox. In case the address denotes a ~~mailing list~~, the incoming mail server sends the message to all subscribers of the list. The incoming mail server also applies ~~filters~~ and generates ~~automatic responses~~, such as ~~delivery failures~~ and ~~out-of-office replies~~.

The incoming mail server waits for connections from mail clients on a different interface. In order to access the mailbox of its user, the mail client has to present appropriate [credentials](#). The user's email address and password are often used to authenticate the client, which is granted unlimited access to the mailbox on success. If the incoming mail server supports [OAuth](#), the mail client can present an [access token](#) to gain potentially limited access to the user's mailbox. The [scopes offered by Gmail](#) are an example of what limited access can look like. While restricted authorization is common for other services, it's not yet the norm for email. Once the client is authenticated, it can retrieve, deposit, and delete messages. It can also mark them as read or flag them for later attention.

▼ Address resolution

How do outgoing mail servers find the incoming mail server of a recipient? As we learned above, an [email address](#) consists of a username and a domain name, separated by the @ symbol. A sender finds the incoming mail server of a recipient by querying the [Domain Name System \(DNS\)](#) for mail exchange (MX) records of the used domain name. If no such records exist, the sender [queries for address records](#) (A or AAAA) of the domain name instead. If the DNS response is not authenticated with [DNSSEC](#), mail might be sent to the server of an attacker. [TLS](#) can prevent this only if the sender requires that the recipient's domain is included in the [server certificate](#), which is ~~usually not the case~~. A standard for ~~securing MX records with TLS~~ exists, though.

A domain can list several servers that handle incoming mail. MX records assign a priority to each incoming mail server. The lower the number, the higher its priority. This is useful for providing redundancy in case the most preferred server is not responding. Several servers with the same priority can be used for [load balancing](#). You can use the following tool to look up the incoming mail servers of a domain you are interested in. It uses an [API by Google](#) to query the Domain Name System and an [API by ipinfo.io](#) to determine the geographic location of each server. The latter is just to remind you that the Internet is a physical infrastructure. Outgoing mail servers need to know only the IP address of the incoming mail server, of course. (A remark on the subdomains you might encounter: [spool](#) is a synonym for [buffer/queue](#), fb probably stands for fallback and alt for alternative.)

Domain:

Query



▼ Null MX record

As we've seen in the [previous box](#), outgoing mail servers fall back to A/AAAA records if no MX records are found at the recipient's domain. If no incoming mail server listens at one of the A/AAAA addresses, an outgoing mail server will attempt to deliver emails to such a domain for days. This is not just a waste of resources, it also delays the ~~bounce message~~ to the sender of the message, who might have simply mistyped the address of the recipient. In order to prevent this from happening, [RFC 7505](#) defines a "null MX record" as 0. similar to how [SRV records](#) indicate the unavailability of a service. You should configure a null MX record on all your domains which neither send nor receive emails.

▼ Dotless domains

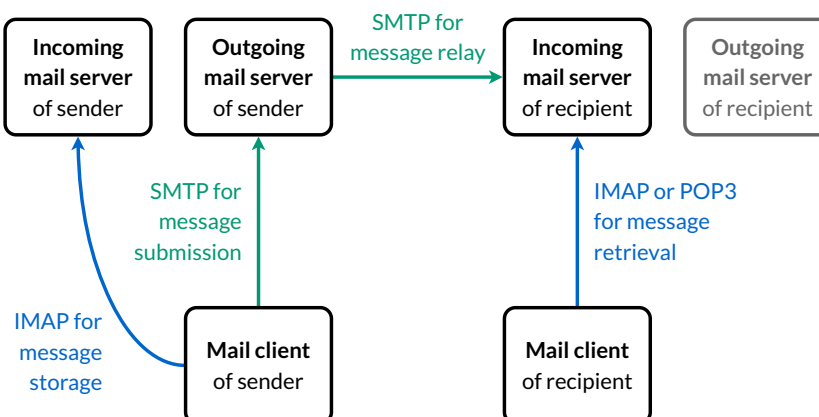
From a technical perspective, top-level domains are domains like any other in the Domain Name System (DNS). This means that they can also have A, AAAA, and MX records and receive mail. Since top-level domains with such records can be used in email and Web addresses without a dot, they are called dotless domains. For example, you can visit <http://ai/> with your browser. [.ai](#) is the country-code top-level domain of Anguilla. The problem with dotless domains is that single labels are often used to address other machines in the local network. Having such names resolve in the global DNS poses a security risk. Additionally, browsers usually pass your input to a search engine if you enter a single word into the address bar. Since dotless domains violate the expectations of users and the assumptions of programmers, ICANN forbids A, AAAA, and MX records on new generic top-level domains since 2013. Out of the 1'502 top-level domains, the following 23 of them have have an A, AAAA, or MX record in April 2021: [.ai](#), [.bh](#), [.cf](#), [.cm](#), [.gp](#), [.gt](#), [.hr](#), [.kh](#), [.lk](#), [.mq](#), [.mr](#), [.pa](#), [.ph](#), [.pn](#), [.sr](#), [.tk](#), [.tt](#), [.ua](#), [.uz](#), [.va](#), [.ws](#), [.xn--11acc](#), and [.xn--mgbah1a3hjkrd](#). (The last two domains are internationalized domain names.) I've determined this list with the script from [RFC 7085](#), which uses IANA's machine-readable list of top-level domains. On yet another note, the formal grammar in [RFC 2821](#) required that the domain part of an email address consists of at least two labels. [RFC 5321](#) no longer has this requirement.

▼ Name collisions

If you run the script from [RFC 7085](#) yourself, you will notice that many name servers cannot be resolved and that a few top-level domains have an A record of 127.0.53.53 and an MX record of 10 your-dns-needs-immediate-attention.{TLD}., where {TLD} is the corresponding top-level domain. The following eight top-level domains have such records in April 2021: [.arab](#), [.cpa](#), [.llp](#), [.politie](#), [.spa](#), [.watches](#), [.xn--mxtg1m](#), and [.xn--ngbrx](#). Since 2014, ICANN requires that new generic top-level domains undergo a controlled interruption for 90 days before becoming operational. Besides the above A and MX records, a controlled interruption also involves a TXT record of Your DNS configuration needs immediate attention see <https://icann.org/namecollision> and an SRV record of 10 10 0 your-dns-needs-immediate-attention.{TLD}.. New country-code top-level domains can but don't have to undergo a controlled interruption. The goal of controlled interruptions is to give IT administrators an opportunity to detect when names which are used only locally suddenly resolve differently than before. This can happen when companies use private top-level domains in their Intranet or when a local DNS resolver extends relative domain names to fully qualified domain names (FQDN) by using search lists. On Unix-like operating systems, a search domain can be configured in the file `/etc/resolv.conf` with a line such as `search example.com`. When the user enters `wiki`, the local DNS resolver might append the search domain to the input and resolve it as `wiki.example.com` only once a query for `wiki` in the global DNS has returned no results. When the top-level domain [.wiki](#) is introduced, the user can no longer access the company's Wiki. Before employees load a resource from an unintended third party or leak information to the Internet, the controlled interruption ensures that the lookup fails and that the name collision can be detected before it causes harm.

Protocols

The above entities communicate with two kinds of protocols: They use delivery protocols to deliver messages and access protocols to access the user's mailbox. As discussed earlier, only SMTP for message relay is mandatory. All other protocols can be replaced in a proprietary setup. For example, there are efforts to combine message submission and mailbox access in a standardized way.



The simplified email architecture with delivery protocols in green and access protocols in blue.

Use of TLS

Historically, SMTP, POP3, and IMAP ran directly on top of the transport layer using the Transmission Control Protocol (TCP), which means that the communication was neither encrypted nor authenticated. Anyone with access to one of the networks through which the communication was routed could therefore read and potentially alter your messages. Even your user password might have been transmitted in the clear. In theory, the

solution is straightforward: Use [Transport Layer Security \(TLS\)](#) to encrypt and authenticate the communication between each pair of entities. In practice, however, you want to be [backward compatible](#): A server that expects requests to be in a specific format cannot suddenly handle a request for a TLS handshake. There are two ways around this problem:

- **Implicit TLS:** Introduce a new port number for each service on which the communication starts directly with a TLS handshake. The protocol variant which uses TLS implicitly is denoted by appending an S to its name. For example, IMAP becomes IMAPS.
- **Explicit TLS** or **STARTTLS**, sometimes mistakenly called **opportunistic TLS**: Allow the client to upgrade an insecure connection to a secure connection with a command once the server has indicated that it supports TLS. The communication is secured only if the client requests this explicitly. The server cannot require the upgrade to TLS as this would break backward compatibility.

With one notable exception, most longstanding email protocols were adapted to support both Implicit TLS and Explicit TLS.

▼ Implicit TLS versus Explicit TLS

When comparing the two approaches, Implicit TLS is significantly easier to implement, debug, and deploy than Explicit TLS. For example, many implementations of Explicit TLS allowed an attacker to [inject commands during the unencrypted phase](#), which would then be executed during the encrypted phase of the protocol. Implicit TLS was once discouraged in favor of Explicit TLS for the [following reasons](#):

- **Implicit TLS leads to new protocols:** The discovery of whether TLS is supported should be made by the client and not by the user, who is likely confused by additional protocol options. However, the same can also be accomplished with Implicit TLS.
- **TLS can be used insecurely:** Unless prohibited by the client or the server, TLS can be used in deprecated versions or with weak security parameters. The protocol variant with Implicit TLS can possibly mislead users into a false sense of security.
- **Worse opportunistic mode:** If the client prefers to proceed without encryption and authentication rather than aborting the connection when the server doesn't support TLS, Implicit TLS forces the client to wait for a timeout on the new port before establishing another connection on the traditional port. Once a secure connection could be established, though, the client should no longer accept insecure connections. Since the insecure protocol could still advertise when its secure variant is available, having only Implicit TLS wouldn't cause a lot of overhead in practice.
- **Port number exhaustion:** If every protocol requires two ports (one to be used with TLS and one without TLS), only half as many protocols can be accommodated in the limited space of port numbers. Luckily, this won't be a problem anytime soon.

Since the ease of deployment should trump any other concerns when it comes to security, [RFC 8314](#) recommends Implicit TLS over Explicit TLS for IMAP, POP3, and SMTP for message submission since 2018. When used opportunistically, Implicit TLS and Explicit TLS provide security only against [passive attacks](#), where an attacker can merely eavesdrop on your communication but cannot interfere with it. In the presence of an [active adversary](#), who can modify and drop network packets, neither Explicit TLS nor Implicit TLS are secure unless the client has a trusted way to know that the server supports TLS. In the case of Implicit TLS, the attacker just has to drop the client's communication to the new port, which forces the client to connect to the old port using the insecure protocol in order to remain backward compatible. In the case of Explicit TLS, the server lists TLS among its capabilities while the communication is not yet authenticated. The attacker can simply [strip TLS](#) from the server's capabilities, which leaves the client with no other option than to continue in plaintext. Alternatively, a client can sacrifice compatibility and refuse to exchange messages over an insecure channel. However, such a change is difficult to introduce because users hate it when their setup no longer works. It is therefore better if the client has a trusted way to know whether the server supports TLS. The following three methods are used in practice to inform the client:

- **Previous connections:** Once a mail server has been upgraded to support TLS, it almost certainly won't be downgraded again. Based on this heuristic, the client can refuse plaintext connections to any server to which it had a TLS connection in the past.
- **Authenticated channel:** While the server cannot reliably inform clients about its capabilities over a downgradeable protocol, it can use another, already authenticated protocol, such as [DNSSEC](#), to convey this information to them.
- **User configuration:** Last but not least, the user can configure the client according to some documentation, which has to be trustworthy, of course. The server's capability might be printed on a leaflet or mentioned on a website secured with HTTPS.

Since securing email deserves much more attention, I've dedicated a whole section to [transport security](#) later in this article.

▼ TLS settings in mail clients

If you have to [configure your mail client](#) manually, it will likely choose the right security option automatically based on the [port number](#) that you've entered. Different clients call the two options differently. Just make sure that one of the two is enabled.

Mail client	Name for Implicit TLS	Name for Explicit TLS
<u>Apple Mail</u>	TLS/SSL	TLS/SSL
<u>Microsoft Outlook</u>	TLS	STARTTLS
<u>Mozilla Thunderbird</u>	SSL/TLS	STARTTLS

Mail clients often use other names for Implicit TLS and Explicit TLS.

As far as I can tell, Apple Mail doesn't distinguish between Implicit TLS and Explicit TLS. As long as the default ports are used, this seems like a reasonable simplification. However, how does Apple Mail determine whether to use Implicit TLS or Explicit TLS when one of the services is deployed on a custom port? Will it try both and see which one worked?

Anyway, we can just hope that mail clients refuse insecure connections when the appropriate TLS option is enabled. I assume this is the case but having the actual behavior documented would still be nice. For example, Apple Mail has an option to allow insecure authentication under "Advanced IMAP Settings", which doesn't disable the "Use TLS/SSL" checkbox as seen below. The [documentation](#) says: "For accounts that don't support secure authentication, let Mail use a non-encrypted version of your user name and password to connect to the mail server." What does this mean? Are they talking about CRAM (challenge-response authentication mechanism), which uses a hash function and not encryption, or does this option make TLS opportunistic? 🤔

The server settings in Apple Mail when "Automatically manage connection settings" is disabled. Also somewhat disappointingly, Apple Mail uses Explicit TLS rather than Implicit TLS by default.

▼ Encryption on the Web

Historically, your web browser used the [HyperText Transfer Protocol \(HTTP\)](#) to fetch websites and other resources from web servers. Just like the original email protocols, HTTP runs directly on top of TCP, which means that its communication is neither encrypted nor authenticated. Since anyone on your network can read the transmitted messages and [hijack your session](#), HTTP should no longer be used. Also similar to the email protocols, there is a variant of HTTP called HTTPS, which uses Implicit TLS to protect your communication. In order to remain backward compatible, HTTPS has to use a different port. While the default port for HTTP is 80, the default port for HTTPS is 443.

What is less well known because it's rarely used, is that HTTP supports Explicit TLS as well. Since version 1.1, HTTP has an [Upgrade header field](#) to upgrade an insecure connection to a secure one. Because Explicit TLS maintains backward compatibility, it can be offered on port 80 as documented in [RFC 2817](#).

▼ Deployment statistics

What percentage of email is encrypted in transit? Interestingly, [Google publishes statistics about this](#) with the data going back as far as December 2013. While not necessarily representative of overall email traffic, the data shows that TLS usage for emails sent from Gmail increased from around 40% in 2013 to around 90% in 2020, while TLS usage for email sent to Gmail increased from around 30% in 2013 to almost 95% in 2020. This rapid increase in [transport security](#) is likely due to the [Snowden effect](#), which sparked initiatives such as [HTTPS Everywhere](#) and [STARTTLS Everywhere](#). Solely relying on TLS for security, including the protection of passwords, has its [own problems](#), though. (You might also be interested in a similar report by Google about [HTTPS usage on the web](#).)

Port numbers

Every protocol specifies a [default port](#) on which servers listen for incoming requests. Instead of scattering the port numbers used by various email protocols throughout the following subsections, here is a table with all the relevant information for future reference:

Protocol	Port for Implicit TLS	Port for Explicit TLS
SMTP for Submission	465	(587)
SMTP for Relay	-	25
POP3	995	(110)
IMAP	993	(143)
IMAP via HTTPS	443	-
ManageSieve	-	4190

The port numbers used by the various email protocols.
Since [RFC 8314](#), Implicit TLS is the preferred option and cleartext is considered obsolete on the port for Explicit TLS.

▼ Why has SMTP for Relay no port for Implicit TLS?

First of all, we'll talk in a minute about why SMTP is different for [submission and relay](#). The [official argument](#) for why SMTP for Relay has no port for Implicit TLS is that [MX records](#) have no way to indicate which port to use and thus port 25 has to be used. In my opinion, this argumentation is misleading. A more accurate answer is that the outgoing mail server had no secure way to discover whether an incoming mail server supported TLS back then, so opportunistic security was all one could hope for at the time. (Manual configuration isn't an option for relay and DNSSEC was standardized only in 2005 and deployed in 2010.) Since opportunistic TLS is more easily accomplished with Explicit TLS rather than with Implicit TLS, we're stuck with Explicit TLS for message relay to this day, even though incoming mail servers can now indicate their TLS capability in a [secure way](#).

(In a [twist of history](#), port 465 was shortly registered for SMTP for Relay with Implicit TLS in 1997 before it was revoked again in 1998 when [STARTTLS for SMTP](#) was standardized. Since some email service providers began to use this port for message submission with Implicit TLS, port 465 was officially recognized for this purpose in [2018](#).)

Delivery protocols

▼ Submission versus relay

The [Simple Mail Transfer Protocol \(SMTP\)](#) is used for two different purposes: The mail client uses it to submit a message to the outgoing mail server of its user, while the outgoing mail server uses it to relay the message to the incoming mail servers of the recipients. Originally, though, mail servers relayed messages from anyone to anyone. This is called [open mail relay](#). In particular, there was no distinction between outgoing mail servers and incoming mail servers. There were just [mail transfer agents](#), which relayed messages among them. Mail clients connected to mail transfer agents just like other mail transfer agents did and asked them to deliver a given message for them. This approach had two problems:

- **Abuse by spammers:** By routing their mail through relay servers of reputable organizations, spammers made it difficult to block their messages based on their origin. Additionally, a single message to a relay server could have a large number of recipients, which allowed

spammers to exploit the still costly bandwidth of others. However, this also meant that a large number of spam messages were identical, which made them relatively easy to filter out on the receiving side.

- **Unwanted rewriting:** Emails have to be in a certain format and mail servers started rewriting them so that they adhere to the standard as well as to organization-specific policies. However, relay servers are not supposed to modify messages and apparently such modifications caused more harm than good.

For these reasons, [RFC 2476](#) introduced a separation between submission and relay in 1998. From then on, mail clients were expected to submit outgoing messages on port 587 instead of port 25 so that mail servers can handle them differently from relayed messages more easily. The RFC also allowed submission servers to require authentication before accepting a message. In the late 90s, submission was often restricted based on the IP address of the mail client. Allowing submission only from within the organization meant, though, that travelling employees couldn't use the outgoing mail server of their organization. Just a few months later, SMTP was extended with a flexible authentication mechanism, which is still in use today. RFC 2476 also permitted submission servers to modify messages in specific ways with the intention that relay servers would stop doing so. Equally importantly, the separation between submission and relay allowed the mail transfer agent of an organization to reject all messages which were addressed to non-local users. This is how the modern email architecture with a server for outgoing mail and a server for incoming mail was born.

As a consequence of this separation, the original SMTP was split into two protocols: One for submission and one for relay. Apart from using different port numbers, they differ mostly in their use of SMTP extensions. The submission protocol is specified most recently in [RFC 6409](#), which also defines what a submission server has to do, what it should do, and what it may do. These aspects affect only how a submission server is supposed to behave but not how mail clients communicate with the server. This is why the two protocols are rarely distinguished when talking about SMTP. For example, Wikipedia also has just a single article for the two protocols. When a distinction is required, such as in technical documents, the submission protocol is called SUBMISSION (or SUBMISSIONS when Implicit TLS is used), while the relay protocol kept the name SMTP. For the reason we discussed above, SMTPS doesn't exist. This doesn't stop Wikipedia from having an article about it, though. By now, you should also understand why the identifier used for the autoconfiguration of a mail client is `_submission` and not `_smtp`.

▼ Header fields and body

We'll have a closer look at the format of messages in the next section but, because we already want to transmit messages in this section, we have to cover the basics now. A message consists of several header fields and an optional body, which follows after an empty line. Each header field has to be on a separate line but can, if necessary, span several lines. Identical to HTTP, header fields are formatted as `Name: Value`. What follows is a simple example message. You can find more examples in [RFC 5322](#).

```
From: Alice <alice@example.org>
To: Bob <bob@example.com>
Cc: Carol <carol@example.com>
Bcc: IETF <ietf@ietf.org>
Subject: A simple example message
Date: Thu, 01 Oct 2020 14:56:37 +0200
Message-ID: <unique-identifier@example.org>
```

Hello Bob,

I think we should switch our roles.
How about you contact me from now on?

Best regards,
Alice

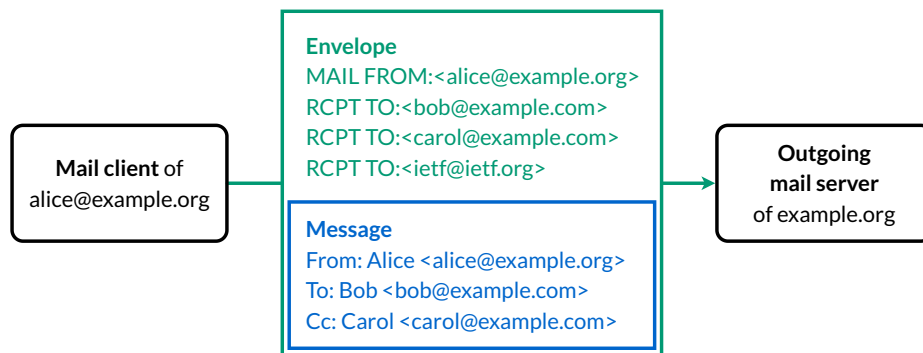
A simple example message with a sender, three recipients, a subject, a date, a message ID, and a body.

▼ Message versus envelope

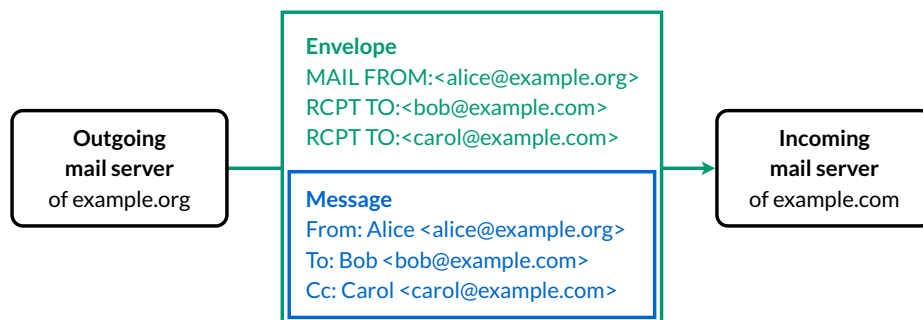
While outgoing mail servers may add missing header fields and sign each message, incoming mail servers should only add trace information to the top of a message and leave the message as is otherwise. The information relevant for handling the message, such as the addresses to deliver the message to and the address to report failures to, belongs to the so-called envelope. The envelope is specific to the Simple Mail Transfer Protocol (SMTP) and it can change completely during the delivery of a message. The message, on the other hand, mostly stays the same during delivery and its format is also used by two access protocols. The important thing to remember is that emails are delivered based on the addresses in the envelope and not the addresses in the header section of the message. Somewhat unfortunately, the fields in the envelope are called similarly to some header fields in the message: MAIL FROM for the address to report failures to and RCPT TO for each address to deliver the message to.

▼ Diverging envelope example

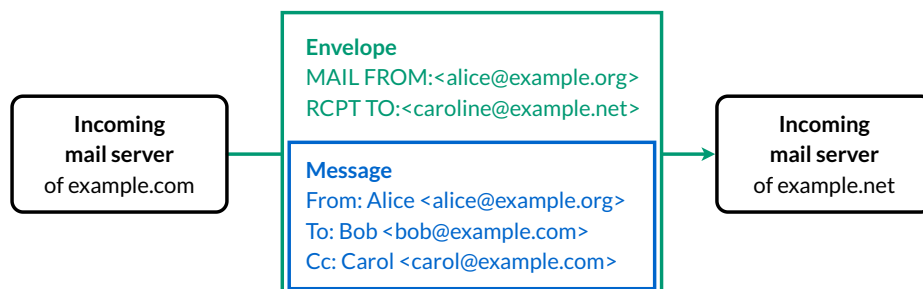
Let's have a look at how the ~~above message~~ is delivered in order to understand how the envelope addresses diverge from the message addresses:



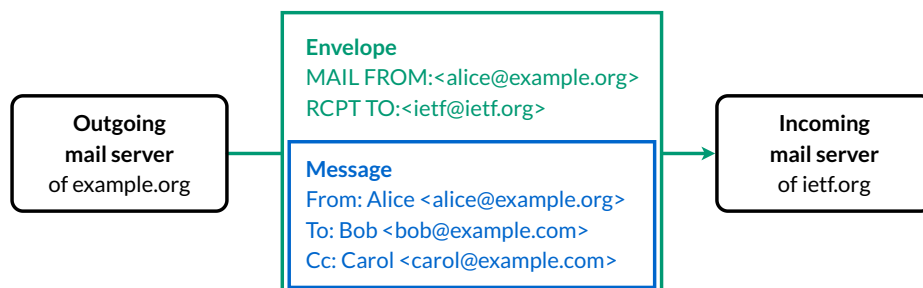
Submission: The mail client of Alice ~~removes the Bcc header field~~ from the message and submits the message with all recipient addresses in the envelope, including the ones of Bcc recipients, to the outgoing mail server. ~~Automatic responses~~ shall be sent to the mailbox of Alice.



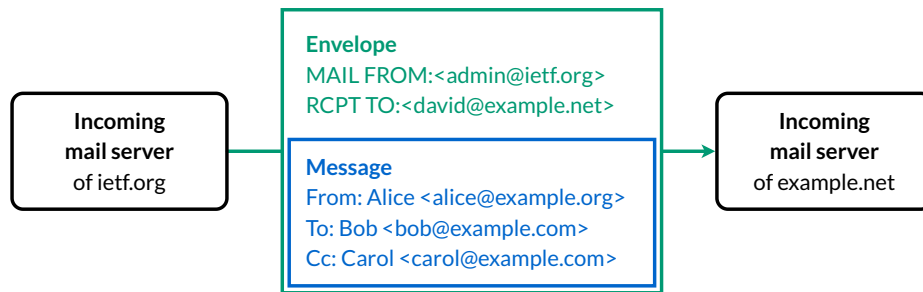
First relay: The outgoing mail server is now responsible for delivering the message to the recipients that the mail client specified. It sees that two recipient addresses are handled by the same domain and delivers the message in a single envelope to the incoming mail server of this domain. The outgoing mail server could also connect to the incoming mail server twice, delivering the message once for Bob and once for Carol. I don't know which approach is more common in practice. [RFC 5321](#) just says that, when the same message is delivered to multiple recipients in the same session, it should be delivered with a command sequence of MAIL FROM, RCPT TO, RCPT TO, DATA rather than MAIL FROM, RCPT TO, DATA, MAIL FROM, RCPT TO, DATA. We'll discuss how the envelope corresponds to protocol messages ~~soon~~.



Alias: The incoming mail server of `example.com` knows that `carol@example.com` is an ~~alias~~ for `caroline@example.net`. It thus forwards the original message without any modifications to the incoming mail server of `example.net`. A potential ~~delivery failure~~ is still reported to Alice.



Second relay: The outgoing mail server of Alice also has to deliver the message to the recipient `ietf@ietf.org`, so it does that.



Mailing list: It turns out that `ietf@ietf.org` is a mailing list. It is now the task of the mail server of `ietf.org` to deliver the message to all subscribers of this list, with one of them being `dave@example.net`. [RFC 5321](#) requires that the bounce address as specified in the MAIL FROM field of the envelope is changed to the entity who administers the mailing list. The entity can be a person but is typically a piece of software, which keeps track of delivery failures in order to revise the list. The RFC also demands that the From field in the message remains the same. [Mailing list tools](#) often modify the message in some ways, for example by adding a field to the header and a footer to the body in order to let recipients of the message unsubscribe from the mailing list. Alias addresses and mailing lists cause difficulties for domain authentication.

▼ Who removes the Bcc header field?

Is removing the Bcc field the job of the mail client or the job of the outgoing mail server? The relevant standards are silent on this but [experts agree](#) that the software which constructs the envelope from the message is responsible for this. If SMTP is used for submitting the message to the outgoing mail server (rather than using one of the custom approaches), the mail client has to remove the Bcc field for the primary (To) and secondary (Cc) recipients. Since this is not clearly stated in the standard, there existed (and maybe still exist) mail clients which relied on the outgoing mail server to remove the Bcc field. However, [RFC 6409](#) lists Bcc removal neither among the mandatory actions nor among the permitted message modifications for outgoing mail servers. While some outgoing mail server software, such as [Postfix](#), which is deployed on around 34% of the reachable mail servers on the Internet, drop the Bcc header field by default, others, such as [Exim](#), which is deployed on around 57% of the reachable mail servers on the Internet, do so only if they are invoked with the -t option. (This option was introduced for use in pipelines, such as `cat message | sendmail -t`.) As a result, users could end up with the list of Bcc recipients going through to non-Bcc recipients depending on their specific combination of mail client and outgoing mail server software. Since neither mail clients nor outgoing mail servers document how they treat Bcc recipients, you have to send a test email to figure out the behavior of your particular setup.

[RFC 5322](#) allows four different behaviors when it comes to Bcc recipients, which we'll study on the basis of another example:

```
From: Alice <alice@example.org>
To: Bob <bob@example.com>
Bcc: Carol <carol@example.com>, David <david@example.net>
Subject: Followup to previous message
Date: Thu, 01 Oct 2020 15:04:26 +0200
Message-ID: <another-identifier@example.org>
```

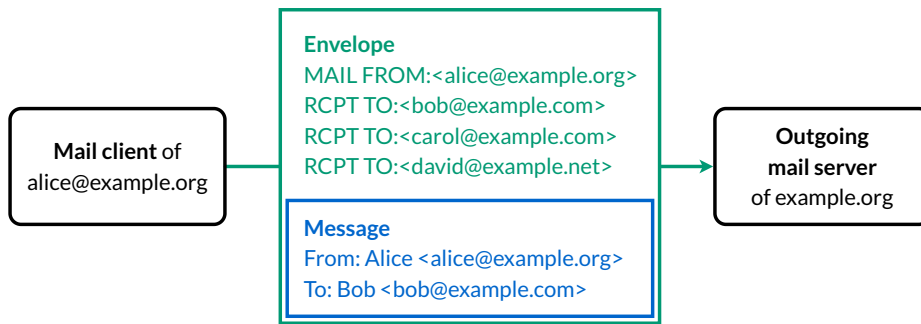
Hi Bob,

I've changed my mind. Please forget the previous message.

All the best,
Alice

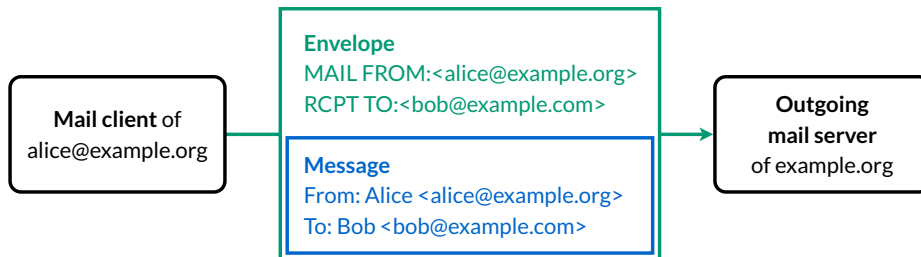
Another example message with several Bcc recipients.

- **Complete removal:** The mail client removes the Bcc field from the message and delivers the message with a single envelope for all recipients to the outgoing mail server. We already encountered this behavior in the [previous box](#). As far as I can tell, this is by far the most common behavior in practice.

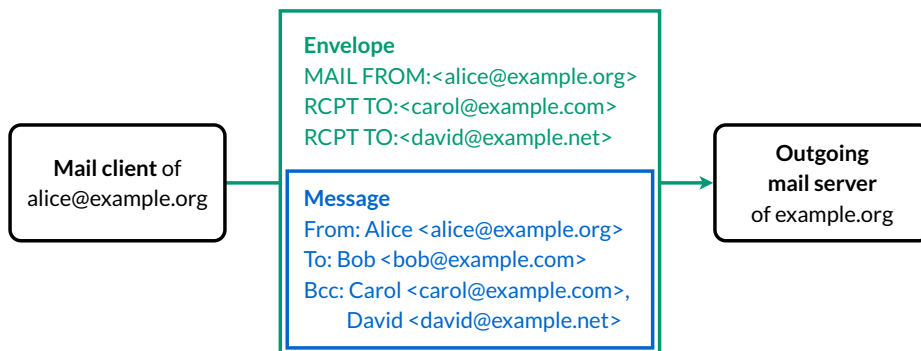


The Bcc field is removed from the message for all recipients.

- **Grouped delivery:** The mail client splits the recipients into two groups. The non-Bcc recipients get the message in which the Bcc field is removed, while the Bcc recipients get the original message, in which all Bcc recipients are listed.

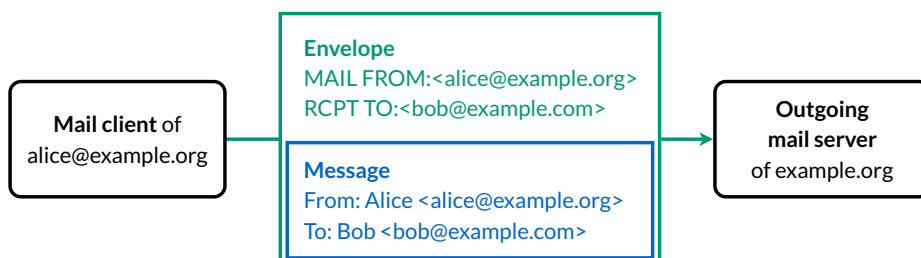


The non-Bcc recipients get the redacted message.

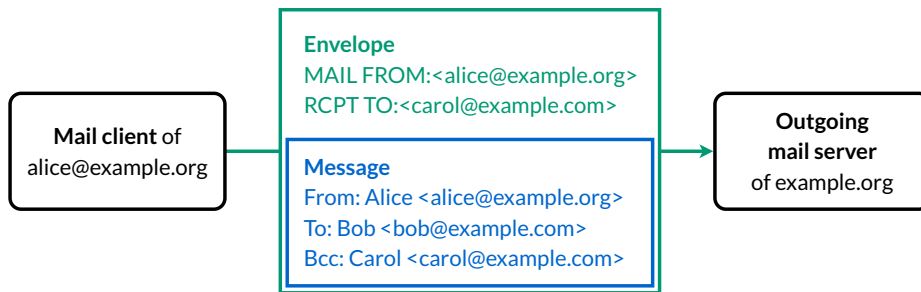


The Bcc recipients get the original message.

- **Individual delivery:** While all non-Bcc recipients receive the same message, each Bcc recipient receives a separate version of the message, in which only they are listed as a Bcc recipient. Just like the first approach, this prevents Bcc recipients from learning about any other Bcc recipient.



All non-Bcc recipients get the same redacted message.



Carol in Bcc gets her own version of the message.



And so does David.

- **Empty field:** While the standard requires that Bcc recipients are never disclosed to non-Bcc recipients, it allows the sender to indicate with an empty Bcc field that there were hidden Bcc recipients. Such a hint can be provided in any of the other three approaches. Therefore, this is more of a second dimension rather than a fourth option, increasing the overall number of Bcc possibilities to $3 \times 2 = 6$.



An empty Bcc field indicates that there were hidden recipients without disclosing them.

The advantage of removing the Bcc field completely is that the mail client has to submit the message only once to the outgoing mail server. The disadvantage of this approach is that Bcc recipients don't learn why they have received a given message: They might have been a hidden recipient or one of the non-hidden addresses might have forwarded the message to their mailbox. Hidden recipients shouldn't send a response to non-hidden recipients because this discloses the fact that they also received the message, which is what the author of the initial message tried to keep secret. In my opinion, mail clients should warn users when they click on "reply to all" for emails that weren't addressed to them, but none of the mail clients I tested did. Even if the Bcc field is removed by the sender, mail clients could deduce from the added trace information whether the message was first received for a listed recipient before being forwarded to their mailbox if the address of the mailbox is not among the recipients. In other words, the drawback of the complete removal approach could be compensated by mail clients but none of them do.

The only way to be sure that a Bcc recipient won't reply to all recipients by accident is to first send the message to the non-Bcc recipients and then forward the message to the hidden recipients. If the hidden recipients don't need to be hidden from each other, you can list them in the To field of the forwarded email. Otherwise, keep them in the Bcc field.

The Bcc field is often used to send an email to undisclosed recipients: The primary recipients of the message are put into Bcc in order to prevent them from seeing each other. Some mail clients, such as the Gmail web interface, indicate this as the sender by using an empty group construct, such as undisclosed-recipients:;, in the To field. As we learned above, this behavior isn't guaranteed by the standard. Given how prevalent it is to use Bcc for undisclosed recipients, I think a new iteration of [RFC 5322](#) should reflect user expectation and formally deprecate the grouped delivery approach unless the user agreed to this behavior.

While the individual delivery approach is nice in theory because recipients are informed about why (and to which alias) they received a message, it isn't ideal in practice because it shifts work from the server back to the client. One of the reasons why we use outgoing mail servers is that mail clients have to submit a message addressed to many recipients only once. Creating and uploading an individual version of the message for each Bcc recipient on the client-side defeats this purpose. While some of the approaches to solve the double submission problem can alleviate this issue especially when large attachments are involved, a simple SMTP extension for submission would do the trick. Since outgoing mail servers have no standardized way to indicate to mail clients that they remove the Bcc header field from the message against the intention or at least spirit of the standard, mail clients might upload individual versions of the message for Bcc recipients in vain. As a consequence, mail clients should opt for complete Bcc removal by default. However, they could do much more to recover some of the lost information on the receiving end and then display this information to their users. If you know about a free mail client which does this, please let me know.

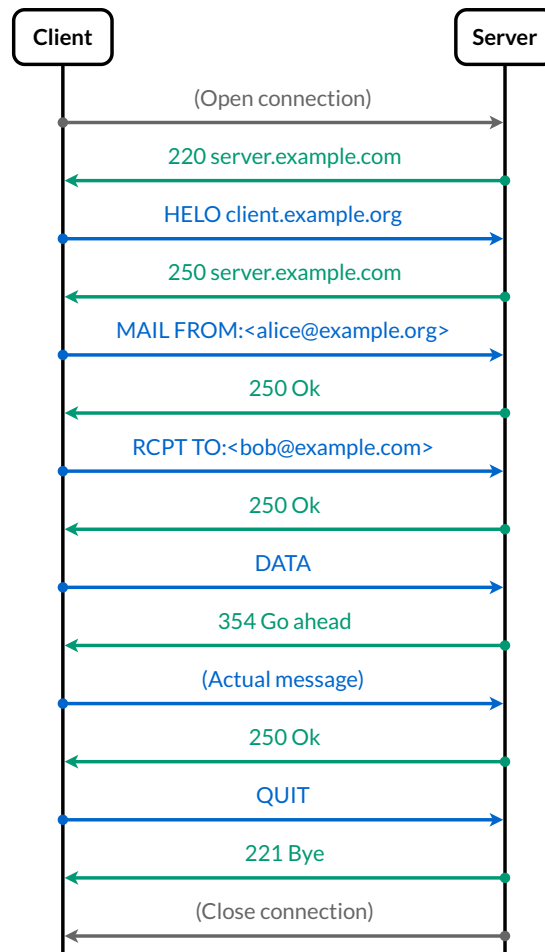
Sometimes, the Bcc field is simply used to prevent certain recipients from getting replies rather than to hide them from other recipients. An example of this is when you move the person who introduced you to someone else to Bcc while still thanking them for the introduction in the reply. This use case could also be addressed with a Do-Not-Reply-To field, which lists all addresses that should be skipped in a reply. Such a header field would also solve the no-reply problem. However, it's almost impossible to bring innovation to email because first implementations and then users would have to adopt such a change.

▼ How does Gmail recover the Bcc header field of sent messages?

The Bcc field serves yet another purpose: It reminds the author to whom they sent a message. While mail clients should remove the Bcc field when submitting a message to an outgoing mail server, they store the message with the Bcc field in the sent folder of the user's mailbox. As you might remember, Gmail does things differently, though. Instead of letting the mail client submit a copy of the message for the sent folder, the outgoing mail server stores all sent messages in the user's mailbox automatically. This leads to the following question: If mail clients remove the Bcc field from a message before sending it, does Gmail recover the Bcc field for the user's copy in the sent folder? The answer is yes. I tested this by submitting messages manually with the tool below. Gmail adds any RCPT TO addresses from the envelope which are not among the recipients of the message to a new Bcc field at the very top of the message (even above the Received and Return-Path header fields, which emails synchronized via IMAP don't have). A consequence is that the display names of Bcc recipients cannot be recovered. This procedure works reasonably well as long as the mail client submits the message only once with all recipients in the envelope. If the mail client opts to deliver a separate version of the message to Bcc recipients, Gmail fails to merge the Bcc recipients from the second submission into the message from the first submission. It just ignores the second message with additional Bcc recipients and the same Message-ID even when it is submitted in the same session (by continuing with another MAIL FROM command after submitting the DATA). If you think you can use this to bypass Gmail's sent archive, I must disappoint you: If you send another message with the same Message-ID but a different body, then Gmail stores the second message in the sent folder as well. Moreover, Gmail always removes the Bcc field, no matter whether you submit the message via SMTP or the web interface.

Simple Mail Transfer Protocol (SMTP)

The Simple Mail Transfer Protocol (SMTP) was first specified in RFC 821 in 1982. As its name suggests, it is a fairly simple protocol:



After opening a TCP connection on port 25, the client sends commands and the server responds with status codes. Once they greeted each other, the client transmits the envelope, followed by the DATA command and the message.

▼ Command syntax

The first question that came to your mind after reading the above sequence diagram probably was: Is HELO a typo? No, it's not. SMTP commands simply consist of four characters. They are almost always written in uppercase, even though they are case insensitive. But yes, HELO does stand for "hello". The purpose of this command is for the client to identify itself to the server with a domain name or an IP address. The identity provided by the client is relevant only in rare circumstances.

Why are the MAIL FROM and RCPT TO commands longer than four characters, then? They're not. The commands are just MAIL and RCPT. FROM and TO denote the subsequent parameter value. Some ESMTP extensions define additional parameters for the MAIL command. The name and value of these additional parameters are separated by an equals sign rather than a colon, though.

▼ Field terminology




Historically, the client could also specify how the message shall be routed. For this reason, the MAIL FROM address is also known as the *reverse path* and the RCPT TO address is also known as the *forward path*. Alternative names for the MAIL FROM address are *bounce address*, *return path*, *envelope from*, and *5321 from* (according to the most recent RFC for SMTP). I will stick to MAIL FROM and RCPT TO for the envelope fields and to From, To, Cc, and Bcc for the message fields. As we will see later on, the MAIL FROM address is added to the message in a Return-Path field. Return-Path is thus a message field rather than an envelope field.

Extended Simple Mail Transfer Protocol (ESMTP)

A framework for extending SMTP was introduced in RFC 1425 in 1993. The extensible protocol, which is backward compatible with SMTP, is called the Extended Simple Mail Transfer Protocol (ESMTP). ESMTP was revised in RFC 1651 (1994), RFC 1869 (1995), RFC 2821 (2001), and most recently in RFC 5321 (2008). The basic idea behind ESMTP is that the client greets the server with the "extended hello" command EHLO instead of the old "hello" command HELO. This indicates to the server that the client understands ESMTP. The server responds with all the SMTP extensions it supports. For the rest of the session, the client can then make use of the server's advertised capabilities.

ESMTP tool

Let's put theory into practice. The following tool generates the command sequence to submit or relay an email with parameters of your choice. One way of using the tool is simply to observe how parameter changes affect the protocol flow. The reason why I built this tool, however, is that you can copy the commands to your command-line interface and send messages without the assistance of a mail client. Since you shouldn't enter your email password on a random website like this one, I recommend that you use the mode for submission only with demo accounts which you've created for this purpose. The password is stored in the local storage of your browser without any protections until you erase the history. Having said that, the tool is open source like the rest of this website and if you don't trust me that this website is served from those files, you can also build and run this website locally. The tool uses Thunderbird's database and Google's DNS API to resolve the server you want to connect to and the API by ipinfo.io to determine your IP address when you click on Determine next to the Client field. The text in gray mimics what the responses from the server likely look like. What you actually receive from the server will be different. As long as the returned status code starts with a 2 or a 3, you should be fine. If the returned status code starts with a 4 or a 5, something went wrong. I list some ideas for things you can try out after the tool. The boxes after that provide you with more information on various aspects, which are useful for troubleshooting problems you might run into. If you need more help, send me an email (probably with your mail client rather than with this tool). 😊

Mode:	Submission ▾	From:	Alice <alice@example.org>
Security:	Implicit TLS ▾	To:	Bob <bob@example.com>
Recipients:	All ▾	Cc:	Carol <carol@example.com>
Domain:	example.org ▾	Bcc:	Dave <dave@example.net>
Server:	submission.example.org		
Port:	465		
Client:	localhost	Determine	
Pipelining:	<input type="checkbox"/>		
Username:	Full address ▾		
Credential:	Plain password (PLAIN) ▾		
Password:	Your password		
Challenge:	CRAM-MD5 challenge from the server		
		Content:	text/plain ▾
		Body:	Hello, It's me. Again.
			  

```
$ openssl s_client -quiet -crlf -connect submission.example.org:465
220 submission.example.org ESMTP Implementation
EHLO localhost
250-submission.example.org at your service, localhost
250-ENHANCEDSTATUSCODES
250 AUTH PLAIN
AUTH PLAIN AGFsaWNlQGV4YW1wbGUub3JnAA==
235 2.7.0 Authentication successful
MAIL FROM:<alice@example.org>
250 2.1.0 Ok
RCPT TO:<bob@example.com>
250 2.1.5 Ok
RCPT TO:<carol@example.com>
250 2.1.5 Ok
RCPT TO:<dave@example.net>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
From: Alice <alice@example.org>
To: Bob <bob@example.com>
Cc: Carol <carol@example.com>
Subject: Yet another message
Date: Thu, 13 May 2021 12:19:37 +0200
Message-ID: <unique-identifier@example.org>

Hello,

It's me. Again.

Alice
.
250 2.0.0 Ok
QUIT
```

Tool instructions

1. Create a new account at an email service provider of your choice. If you opt for Gmail, you should read [this box](#) first.
2. Enter the address of your account in the From field and your password in the Password field. Set the Mode to Submission.
3. After composing the message (To, Subject, and Body), try to submit it to the outgoing mail server with the listed commands.
4. The first line opens a TLS channel to the specified Server. All other commands are sent to the server inside this channel.
5. You can copy each line in bold to your [clipboard](#) by clicking on it, which includes the newline character to [submit the command](#).
6. If the mail was submitted successfully, you can add more To or Cc recipients. By copying only some of the generated RCPT TO commands but the full message, you suppress the delivery of the message to the skipped recipients. For those that receive the message, it looks as if the message was delivered to all the recipients in the message. I already mentioned this problem [above](#).
7. Besides faking recipients, you can also try to fake the sender. Switch the mode from Submission to Relay and change the From field to an address that you don't own. Now try to send the message directly to the incoming mail server of one of the recipients. If the incoming mail server and the domain which you try to send the email from are [properly configured](#), your message should make it at best into the spam folder of the recipient. Chances are that your message will be rejected during the SMTP session or silently dropped thereafter. The incoming mail server might also [graylist](#) or [blacklist](#) your IP address. Since you usually don't relay email from your computer, this is nothing to worry about. Forging the sender address is known as [spoofing](#). Be careful which domains you try to impersonate. If the domain owner configured a [DMARC record](#), they might be informed about your spoofing attempt and even receive the [content of your message](#).

Important: Be a nice person and don't scam others! If you spoof the sender of an email in bad faith, you likely commit a crime in most countries. I showed you this attack for educational purposes only because I believe that seeing is believing. We can improve the state of email security only if consumers start demanding better security. In this spirit, I encourage you to relay spoofed emails only to your own mailbox. If such a spoofed email lands in your inbox, ask your email service provider to be more rigorous in filtering scam emails or use the service of a different provider. You're hopefully also more motivated now to read the rest of this article. In short, have fun with the above tool but always remember that with great power comes great responsibility!

Tool explanations

▼ Command-line interface

If you've never used the [command-line interface](#) of your operating system before, I suggest that you read a proper introduction first. If you have no clue about what you're doing, it's easy to mess up your computer. Additionally, you shouldn't blindly execute arbitrary commands from the Internet. Ideally, you should always try to understand what a command does based on a separate source first. Having said that, the default program providing a command-line interface is called [Terminal](#) on macOS. It's located in the /Applications/Utilities folder at the top of your file system. The openssl tool should already be installed. Continue [here](#) to test this. On Windows, the default command-line program is called [Command Prompt](#). Various third parties provide [OpenSSL binaries](#) for Windows. Here is a [guide](#) for installing OpenSSL on Windows 10, which you can follow at your own risk.

▼ Clipboard verification

If a website copies commands to your [clipboard](#) for you, you should verify the content of your clipboard before pasting it into your [command-line interface](#). Otherwise, a malicious website can display one command and copy another command. One way to inspect your clipboard is to always paste its content into a [text editor](#) first. Since this is a hassle, you likely won't do this for long. A better approach is to have a window which displays the current content of your clipboard. On macOS, the [Finder](#) has a "Show Clipboard" command in the "Edit" menu. Unfortunately, this window is visible only if Finder is the active application. A different approach is to open a new window in your [terminal](#) and paste the clipboard once a second with the [watch command](#):

```
# macOS:
$ watch -n 1 pbpaste

# Linux:
$ watch -n 1 xclip -selection clipboard -o

# Windows:
$ powershell -command "while (1) { Clear; Get-Clipboard; Sleep 1 }"
```

How to watch your clipboard in your command-line interface. Press "control c" to exit the program.

▼ OpenSSL versus LibreSSL

[OpenSSL](#) used to be the most important open-source library for TLS functionality. (When OpenSSL was first released in 1998, [TLS was still called SSL](#)). After the [Heartbleed](#) security vulnerability in April 2014, the [OpenBSD](#) project forked [LibreSSL](#) from OpenSSL. In order to remain as compatible as possible, the command-line tool is still called `openssl`. Since [macOS 10.13.5](#), Apple ships LibreSSL and no longer OpenSSL. I mention all of this here only because the arguments of the two commands are no longer identical. Here are the documentations of the `s_client` subcommand for [OpenSSL](#) and for [LibreSSL](#).

Execute the following command to figure out whether `openssl` is installed on your system and which implementation you have:

```
$ openssl version
LibreSSL 2.8.3
```

▼ Common SMTP extensions

The difference between ESMTP and SMTP is that ESMTP allows the server to list extended capabilities, which the client can make use of during the session. Let's have a look at some common SMTP extensions on the basis of what Gmail supports:

```
$ openssl s_client -quiet -crlf -connect smtp.gmail.com:465
depth=2 OU = GlobalSign Root CA - R2, O = GlobalSign, CN = GlobalSign
verify return:1
depth=1 C = US, O = Google Trust Services, CN = GTS CA 101
verify return:1
depth=0 C = US, ST = California, L = Mountain View, O = Google LLC, CN = smtp.gmail.com
verify return:1
220 smtp.gmail.com ESMTP [your session ID] - gsmt
EHLO localhost
250-smtp.gmail.com at your service, [your IP address]
250-SIZE 35882577
250-8BITMIME
250-AUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER XOAUTH
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-CHUNKING
250 SMTPUTF8
QUIT
221 2.0.0 closing connection [your session ID] - gsmt
read:errno=0
```

A transcript of a session with the outgoing mail server of Gmail when using [Implicit TLS](#). [Brackets indicate redacted information.]

As can be seen in the above transcript, Gmail's outgoing mail server supports the following SMTP extensions:

- **SIZE (RFC 1870)**: This extension allows the server to specify an upper limit on the size of messages it accepts in bytes as part of the EHLO response. Gmail apparently accepts messages of almost 36 MB. The extension also allows the client to specify the size of the message in bytes as part of the MAIL command: `MAIL FROM:<alice@example.org> SIZE=1234`. The server can then reject the message [for individual recipients](#) in its response to each RCPT command, for example because a mailbox no longer has enough space to store a message of the stated size. Doing so has the advantage that a large message doesn't even have to be transmitted if it will be rejected for all recipients based on its size. (The declared size can be an [estimate](#).)
- **8BITMIME (RFC 6152)**: MIME stands for [Multipurpose Internet Mail Extensions](#) and we'll discuss this [later](#). SMTP originally required the message to consist of 7-bit [ASCII](#) characters. This extension allows the server to signal that it'll preserve the 8th bit of each byte in the message body. The client can then indicate in the MAIL command that the content of the message contains bytes outside of the ASCII range: `MAIL FROM:<alice@example.org> BODY=8BITMIME`. The server can still enforce a [limit on the length of each line](#), though. Therefore, this extension doesn't enable binary data transfer without encoding.
- **AUTH (RFC 4954)**: This extension allows the server to [authenticate](#) the user in the [submission protocol](#) before accepting a message for relay. Since the [above tool](#) makes extensive use of this extension, it deserves its own [information box](#).
- **ENHANCEDSTATUSCODES (RFC 2034)**: This extension allows the server to respond with more precise [status codes](#) than the ones specified in the [original standard](#). The server indicates that it returns enhanced status codes to the client by listing the extension in its response to the EHLO command. The server then prepends the enhanced status codes to the text part of the original status codes. The structure of enhanced status codes is `class.subject.detail`, with the values specified in [RFC 3463](#) and maintained in a [registry by IANA](#).

- **PIPELINING (RFC 2920):** The goal of this extension is to reduce the number of round trips during an SMTP session. Instead of having to wait for a response from the server after each command, it allows the client to send several commands in a single packet to the server. The standard requires that EHLO, DATA, and QUIT are the last command in a batch of commands. AUTH must also be the last command in a batch unless the authentication method is PLAIN, which makes the command non-interactive. The server then returns all the status codes at once, matching the order of the transmitted commands. The reason why I've implemented pipelining in the above tool is because it makes copying the commands much quicker.
- **CHUNKING (Section 2 of RFC 3030):** This extension allows the client to split the message into several chunks and transfer each chunk separately, which is especially useful for large messages. Instead of the DATA command, the client can send one or several BDAT commands, which are immediately followed by the respective chunk. When using the BDAT command, the client specifies the size of the chunk in bytes, which has the advantage that the client doesn't have to escape lines containing a single period and that the server doesn't have to scan the transmitted data for the {CR}{LF}. {CR}{LF} sequence in order to determine the end of the message. This length prefix turns SMTP into a binary protocol temporarily. The client indicates the last chunk by appending LAST after the chunk size to the BDAT command. The RFC contains a simple example.
- **SMTPUTF8 (RFC 6531):** This extension allows the client to use UTF-8 instead of just ASCII in the MAIL and RCPT commands as well as the message. A server which supports the SMTPUTF8 extension also has to support the 8BITMIME extension. SMTPUTF8 facilitates the internationalization of email addresses.

If you connect to a different server, you likely encounter other extensions as well. The server indicates the end of the response to the EHLO command by using a hyphen after the status code for all but the last line.

▼ Backward compatibility

ESMTP uses the same port as SMTP, so how does ESMTP ensure backward compatibility with SMTP? (Since submission was split from relay in 1998 while ESMTP dates back to 1993, we're talking only about port 25 here.) Remember that when an outgoing mail server connects to an incoming mail server, it assumes the role of the client in that interaction. There are only two cases to consider:

- **Old client → new server:** ESMTP servers still have to accept the old HELO command in order to remain compatible.
- **New client → old server:** SMTP servers which don't understand the EHLO command respond with the error code 500. The client can either QUIT the connection or continue with the HELO command. According to this source, some mail clients send the EHLO command only if the first line from the server, which starts with the status code 220, contains ESMTP. This explains why most servers include ESMTP in their greeting even if the standard doesn't require it.

▼ STARTTLS extension

Explicit TLS is implemented with an extension called STARTTLS, which is specified in RFC 3207. The reason why Gmail didn't list this extension is because we used SMTP with Implicit TLS on port 465. If we open a TCP connection on port 587, it's there:

```
$ telnet smtp.gmail.com 587
Trying 108.177.126.109...
Connected to smtp.gmail.com.
Escape character is '^['.
220 smtp.gmail.com ESMTP [your session ID] - gsmtip
EHLO localhost
250-smtp.gmail.com at your service, [your IP address]
250-SIZE 35882577
250-8BITMIME
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-CHUNKING
250 SMTPUTF8
QUIT
221 2.0.0 closing connection [your session ID] - gsmtip
Connection closed by foreign host.
```

The STARTTLS extension is listed when we connect without TLS to Gmail's outgoing mail server.

If the STARTTLS extension is listed in the response to the EHLO command, the client can ask the server to upgrade the insecure channel to a secure one with the STARTTLS command. If the server responds with the status code 220, the client can continue with the TLS handshake. Once the handshake is completed, the client and the server are reset to their initial state. In particular, the server must forget about the client's argument to the EHLO command, whereas the client must forget about the extensions supported by the server. The client should send

another EHLO command, to which the server can respond with a different list of extensions than before the TLS handshake. For example, the AUTH extension is missing in the above list because passwords of users shouldn't be transmitted over an insecure channel. You can use the following command in your command-line interface to let openssl issue the STARTTLS command after an initial EHLO command and then continue with the TLS handshake:

```
$ openssl s_client -quiet -crlf -starttls smtp -connect smtp.gmail.com:587
```

When using `-starttls smtp`, openssl starts with a TCP connection and upgrades it to a TLS connection by issuing the STARTTLS command.

If the server doesn't list the STARTTLS extension or responds with a status code other than 220 to the STARTTLS command, the client has to decide whether it wants to continue or abort the connection. As explained [earlier](#), neither Explicit TLS nor Implicit TLS is secure against [downgrade attacks](#) when used [opportunistically](#). The belief that only Explicit TLS with STARTTLS has this weakness is a common misunderstanding. Due to backward compatibility, it's up to the client to require a secure channel or to abort otherwise. If the client does require TLS, it might no longer be able to submit or relay messages to some servers, though.

As a side note: [OpenSSL](#) has a [-name option](#) to let you specify the argument to the initial EHLO command. Since the server must forget about this argument after the TLS handshake, I have no idea what's the point of providing this option. This is likely the reason why [LibreSSL](#) doesn't support this option in the first place.

▼ User authentication

In order to protect their reputation and to reduce spam and phishing, outgoing mail servers authenticate their users before accepting messages for relay. This is done with the AUTH extension as specified in [RFC 4954](#). The AUTH extension itself is also extensible: Servers can support new mechanisms, which clients can then make use of. Since many [application-layer protocols](#) require authentication, the IETF community abstracted the various mechanisms into the so-called [Simple Authentication and Security Layer \(SASL\)](#), which is specified in [RFC 4422](#). [IANA](#) maintains a list of [SASL mechanisms](#). SMTP servers list all the mechanisms that they support after AUTH in their response to the EHLO command. We're interested in only four of them:

- PLAIN ([RFC 4616](#)): The client sends the [Base64](#) encoding of the user's username and password as an argument to the AUTH command to the server. The username and password are separated by the [null character](#). If you don't trust the [above tool](#), you can compute the encoding on your command line as `echo -ne '\000username\000password' | openssl base64`. The `echo` command writes the argument to its [standard output](#), which is then [piped](#) to openssl for the Base64 encoding. The `-n` option to echo suppresses the trailing newline in its output and the `-e` option enables interpretation of backslash escapes. The reason why I used four zeros instead of just two in the escape sequence is to not cause any troubles if your username or password starts with a number. And if you're wondering why there is a leading null character: The standard supports an [additional field](#) at the beginning, which is usually left empty in the case of SMTP. The username and password can consist of any Unicode character except the null character. All characters have to be encoded with [UTF-8](#).
- LOGIN ([draft-murchison-sasl-login](#)): This mechanism is obsolete but since it's still widely offered, I decided to implement it in the above tool as well. Instead of sending the username and the password together, the server prompts for them separately once the client has initiated the authentication with AUTH LOGIN. The LOGIN mechanism has the same security properties as the PLAIN mechanism, it just requires more round trips and prevents [pipelining](#) because it's interactive.
- CRAM-MD5 ([RFC 2195](#) and [draft-ietf-sasl-crammd5](#)): As far as I can tell, this mechanism is not widely used by mail servers but still widely supported by mail clients. I cover this mechanism in more detail in a [separate box](#). The summary is that the client puts the password and a challenge from the server through a [one-way function](#) and sends the output of this function to the server instead of the password. This was useful against passive attackers before the widespread deployment of TLS.
- SCRAM ([RFC 5802](#)): SCRAM is not much more complicated than CRAM-MD5 but has [much better properties](#). Unfortunately, it's not widely used so I didn't bother to implement it in the above tool. In my opinion, all weaker [password-based authentication mechanisms](#) should be replaced with SCRAM (or another, similarly secure mechanism). Therefore, it also deserves its [own box](#).

Please note that the tool hides the password in the input field but unless you use CRAM-MD5, anyone who can take a picture of your screen can easily decode the entered password. When authenticating to an SMTP server, the server responds with either `235 2.7.0 Authentication successful` or `535 5.7.8 Error: authentication failed`.

▼ Gmail authentication failure

If you want to submit an email to Gmail with the instructions generated by the [above tool](#), you have to allow access from [less secure apps](#) in your [account settings](#). If the authentication still fails, you might have to complete [this page](#) according to [these instructions](#). Please note that Google disables access from less secure apps automatically if it's not being used for some time.

▼ Reverse DNS entry

I didn't go into much detail about [reverse DNS lookups](#) in my [previous article about the Internet](#). Simply put, IP address ranges [are allocated to Regional Internet Registries \(RIR\)](#), which allocate subranges to regional [Internet service providers \(ISP\)](#). The DNS zones under the special `in-addr.arpa` domain are delegated along the same hierarchy. For example, when the [Internet Assigned Numbers Authority \(IANA\)](#) allocated the IP address block `123.xxx.xxx.xxx` to the [Asia-Pacific Network Information Centre \(APNIC\)](#), it also delegated the DNS zone `123.in-addr.arpa` to APNIC. You can [check this](#) with the [DNS tool](#) below:

Domain: Type: DNSSEC: ☐

When APNIC allocates the IP block `123.234.xxx.xxx` to an ISP, it also delegates the DNS zone `234.123.in-addr.arpa` to this ISP. The ISP can then create so-called pointer records (PTR) to map IP addresses to domain names. While DNS is normally used to resolve domain names to IP addresses, pointer records under `in-addr.arpa` are used to do the reverse. The reason why you have to reverse an IP address when doing a reverse DNS lookup is because in IP addresses the root of the allocation hierarchy is on the left whereas in domain names the root of the delegation hierarchy is on the right. In reality, the situation is a bit more complicated because the 32-bit IPv4 address ranges are no longer just allocated along the byte boundaries but also split at arbitrary positions. This is known as [classless inter-domain routing \(CIDR\)](#) and solved by [classless in-addr.arpa delegation](#).

Since Internet service providers usually don't configure reverse mappings for the IP addresses of their residential customers, incoming mail servers use this [as a heuristic to fight spam](#). If you use the [above tool](#) to relay a message directly to an incoming mail server, your chances of having the message delivered are much higher if your [public IP address](#) has a reverse DNS entry. Somewhat ironically, this means that spoofing emails often works better when you use the Wi-Fi of a hotel or a restaurant instead of your own. If your public IP address has a reverse DNS entry, the tool determines it when you click on "Determine":

Client:

▼ Newline characters

In [teleprinters](#) (printers that operated like [typewriters](#)), moving the carriage, which outputs the characters onto paper, back to the start of the same line and moving the page to the next line were two separate instructions. The former is known as [carriage return \(CR\)](#), the latter as [line feed \(LF\)](#). Both CR and LF were included as [control characters](#) in the [American Standard Code for Information Interchange \(ASCII\)](#). While some operating systems, such as [Windows](#), opted to encode a [newline](#) as a sequence of both CR and LF, other operating systems, such as [Linux](#) and [macOS](#), use only LF to encode a newline. As you can imagine, this causes a lot of [interoperability issues](#). Both [SMTP](#) and the [message format](#) require that lines end with both CR and LF. By using the [-crlf option](#), `openssl` makes sure that this is the case.

▼ Message termination

When using the [DATA command](#), the transmission of the message is terminated by a period on a line of its own. So what happens if you include a line with a single period in an email? [SMTP specifies](#) that the sender has to insert an additional period at the beginning of every line which starts with a period before transmitting the message. The recipient then removes the leading period from every line which has additional characters in order to restore the original message. Periods at the beginning of lines in a message are [escaped](#) like this only for transmitting the message with the DATA command but not when storing the message (or when using the [BDAT command](#)). You don't have to worry about this; the [tool above](#) does the escaping for you.

▼ Origination date

The [Date field](#) indicates the date and time at which the author of the message pushed the "Send" button. It's not supposed to reflect when the message is actually sent, though: If the device is offline when the user clicks on "Send", the message is queued locally and the Date field isn't updated when the message is submitted. Messages [must have](#) a single Date field and outgoing mail servers [may add one](#) if it's missing. The outgoing mail servers that I've checked don't enforce any rules on the Date. I've successfully submitted messages whose Date was one year in the past or in the future. Do mail clients display messages with the date that was chosen by the sender? Most don't, some do. Apple Mail and the [webmail](#) interfaces of Gmail, Yahoo, and Outlook display the date when the message was received, completely ignoring the Date specified by the sender. I assume that they determine the received date based on the uppermost [Received header field](#). Thunderbird and [Postbox](#), on the other hand, display messages with the sender-chosen Date by default. Since sender-chosen means attacker-chosen, I think this behavior is problematic, especially since these mail clients [tell all recipients that you're using them](#). For example, a scammer can backdate financial predictions and reference such messages in a current email. Alternatively, you can backdate an email to meet a passed deadline. Or if you want to make sure that your message lands at the top of the recipient's inbox, you can choose a date in the future. Such tampering is easy to detect if you know how to inspect the [raw message](#). For ordinary users, however, a warning should be shown, in my opinion. I reported [this issue](#) to the Thunderbird team, but they were not interested in addressing it.

▼ Spoofed sender during submission

Can you spoof the sender address not only during relay but also during submission? Or more precisely: Can you authenticate to an outgoing mail server as one user but then use the address of a different user in the MAIL FROM and From fields? According to [RFC 6409](#), outgoing mail servers may enforce submission rights but they don't have to. If you want to know how your email service provider handles such submissions, you have to try it. Some mail clients, such as [Thunderbird](#) and [Roundcube](#), support alternative sender identities to spoof the sender address for you. If you want to do this manually in order to see the response from the server, you can change the From address in the [above tool](#) after you have already copied the AUTH command to your command-line interface. Gmail, for example, accepts submissions with the address of a different user in the MAIL FROM and From fields but then replaces both with the address of the authenticated user and adds the spoofed sender address in an X-Goog-Original-From header field. Mail server software, such as [Postfix](#), needs to be [configured](#) to reject submissions where the sender address doesn't match the authentication address. Postfix also has an [option](#) to add the authenticated sender to the [Received header field](#). I think outgoing mail servers should reject spoofed sender addresses even if there is legitimate use.

I reported to [Gandi.net](#) on 27 October 2020 that their outgoing mail server accepts submissions with spoofed MAIL FROM and From addresses. On the one hand, they told me that some of their customers use alternative sender identities and that they won't enforce any rules for them. On the other hand, they let me know that they would address the issue before my [90-day disclosure deadline](#). When I tested this again before publishing this article, I got the impression that more of my test messages were rejected by [their spam filter](#) but I could still authenticate myself as a Gandi user and then use my Gmail address in the MAIL FROM and From fields. This allows an attacker to abuse the [reputation](#) of Gandi's mail server at least for targeted attacks.

▼ Limitations of the above tool

- The [example domains](#) don't work, you have to replace all example addresses before executing the generated commands.
- The tool supports only the complete removal of Bcc recipients or [grouped delivery](#), but no individual delivery.
- The tool supports only the PLAIN, LOGIN, and CRAM-MD5 [user authentication](#) mechanisms.
- The tool doesn't enforce [line-length limits](#). Break lines longer than 1000 bytes yourself.
- The tool doesn't support any [extensions](#) except STARTTLS, AUTH, and PIPELINING.
- The address format is more restrictive than necessary:
 - No [quoted strings](#) in the local part,
 - No support for the [group construct](#),
 - No support for [folding whitespace and comments](#).
 - Only ASCII characters supported in addresses and [display names](#) (i.e. you have to do [header encoding](#) and [domain encoding](#) yourself).

▼ Other SMTP commands

Besides [EHLO](#), [MAIL](#), [RCPT](#), [DATA](#), and [QUIT](#), there are some other SMTP commands, which are rarely used in practice:

Command	Argument	Description
RSET	-	Reset already transmitted sender, recipient, and mail data.
VRFY	Mailbox	Verify whether the given mailbox exists on the server.
EXPN	Mailing list	Expand the given mailing list (i.e. return the members).
HELP	[Command]	Ask for helpful information about the optional command.
NOOP	-	Do nothing besides keeping the connection alive .

These commands can be used [at any time during a session](#).
VRFY and EXPN are usually disabled for [security reasons](#).

Let's look at two examples:

```
$ openssl s_client -quiet -crlf -starttls smtp -connect spool.mail.gandi.net:25
[...]
VRFY kaspar@ef1p.com
502 5.5.1 VRFY command is disabled
```

What a response to the VRFY command usually looks like. (The reply code of a disabled VRFY command [should be 252](#), though.)

```
$ openssl s_client -quiet -crlf -starttls smtp -connect gmail-smtp-in.l.google.com:25
[...]
HELP
214 2.0.0 https://www.google.com/search?btnI&q=RFC+5321 [your session ID] - gsmtip
```

How Gmail responds to the HELP command. 😊

Automatic responses

In certain configurations, mail servers send a message in response to an incoming message, which leads to the following problems.

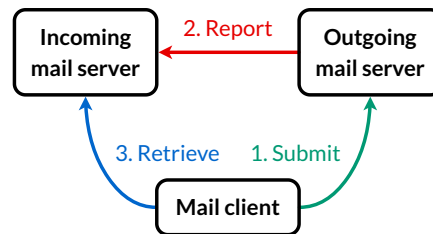
▼ Mail loops

If a received message causes a mail server to send one or several messages which in turn trigger further messages, we end up with a chain reaction. In the case of email, chain reactions get out of control if messages are sent in a loop or if the forwarding rules result in a combinatorial explosion. Both of them can happen by accident or as a denial-of-service attack by an attacker. Depending on the circumstances under which they happen, chain reactions are prevented with the following measures:

- **Automatic responses** are often sent to inform the sender that a message could not be delivered or that the recipient won't read the message anytime soon. Sometimes, automatic responses are used to pose a challenge to the sender which needs to be completed in order for the message to be delivered. The incoming mail server of the sender should not respond to such responses automatically as this could result in messages being sent back and forth indefinitely between the two systems. Automatic responses should always be sent to the MAIL FROM address, which was specified in the envelope of the message. By using an empty MAIL FROM address (MAIL FROM: <>), a sender can indicate that no automatic response shall be sent back. Additionally, automatically submitted messages should be marked with the Auto-Submitted header field, which is specified in RFC 3834. If an automatic process sends a message in response to another message, the value of this header field should be set to auto-replied. If the message is triggered by another event, the value should be set to auto-generated. If a message contains an Auto-Submitted header field with a value other than no, no automatic responses should be sent. Furthermore, Microsoft Exchange Server introduced the custom header field X-Auto-Response-Suppress, allowing the sender to control which types of automatic responses shall be suppressed.
- **Email forwarding** can cause loops as well. If `alice@example.org` was an alias for `alice@example.com` and vice versa, emails would be forwarded in an infinite loop. Since only loops should be prevented but neither message forwarding nor automatic responses, none of the previous techniques can be used. Instead, incoming mail servers add a non-standardized header field, such as `Delivered-To` or `X-Loop`, with the recipient's address to messages before forwarding them. When incoming mail servers receive a message, they can simply go through its header fields to determine whether the message has already been delivered to the specified mailbox. If the message has already been delivered, they respond with a delivery failure. Another way to detect loops is to count the Received header fields in a message. If they exceed a certain threshold, the message is bounced. Both techniques require that mail servers only add additional header fields without removing existing ones.
- **Mailing lists** forward incoming messages to all subscribers of the list. If two mailing lists are subscribed to one another, if automatic responses are sent to the mailing list's address, or if a subscriber automatically forwards messages back to the mailing list, a mailing list is involved in a mail loop. Such a loop can be busted with the same techniques as before: Mailing lists shouldn't forward messages with a header field of `Auto-Submitted: auto-replied` or `Delivered-To` followed by the address of the mailing list. However, mailing lists pose an additional problem: If mailing lists are subscribed to one another, the number of combinations before a loop occurs explodes with the number of involved mailing lists. For example, if you're subscribed to ten mailing lists which are all subscribed to one another, a single message to one of them results in almost one million messages delivered to your inbox. To prevent this, mailing lists shouldn't forward messages from other mailing lists, which can be detected with List-* header fields, such as `List-Id` or `List-Unsubscribe`.

▼ Bounce messages

When a mail server fails to deliver a message, it should send a so-called bounce message to the sender to notify them about the failed delivery. Since bounce messages are automatic responses, they must be sent to the MAIL FROM address of the envelope.



How the user learns about a delivery failure.
If the delivery of a message fails on the recipient's side,
the bounce message (in red) is generated by a different system.

Historically, bounce messages were in a format that could be interpreted only by a human sender. However, many messages are sent by automated systems, which should also be able to detect when a message couldn't be delivered. For example, mailing list software should be able to remove no longer valid addresses from the list automatically. Two techniques address this problem:

- **Machine-processable non-delivery reports (NDR):** RFC 3464 specifies how multipart messages can be used to send so-called delivery status notifications (DSN) to the sender in a standardized way. In short, the bounce message is marked with Content-Type: multipart/report; report-type=delivery-status; boundary="..." and the machine-processable part is labeled with Content-Type: message/delivery-status. The report contains message-specific and recipient-specific fields, which are separated by a blank line. The RFC includes some examples. The advantage of this approach is that even mail clients can make use of the report. Its disadvantage is that not everyone supports this format and even if everyone did, the sender doesn't learn for which recipient the message couldn't be delivered if it was forwarded by an alias address. Since non-delivery reports include the header fields of the original message, this could be recovered from the trace information.
- **Variable envelope return path (VERP):** Since the MAIL FROM address of the envelope can be different from the From address of the message, it can encode the recipient's address. For example, when the mailing list server at list@example.com sends a message to alice@example.org, it can choose the MAIL FROM address as list-owner+alice=example.org@example.com. Since it has to be a valid address, the @ of the recipient's address has to be replaced with something else, such as =. As long as the mailing list software can access the automatic responses that were delivered to such addresses, it can easily associate a response with the recipient who sent it. The software needs to guess only whether the response denotes a failed delivery or an out-of-office reply. It should remove addresses from the mailing list only if messages to a particular recipient cannot be delivered over a period of several weeks. If you look at the Return-Path header field of messages sent by mailing list providers, such as Mailchimp, you see an address which identifies you. The good thing about VERP is that it works very reliably. On the downside, mail clients can make use of this technique only with subaddressing. Since the syntax for this is specific to each email service provider if subaddressing is supported at all, I'm not aware of any mail clients which use VERP to enhance the user experience of delivery failures. Moreover, VERP requires that the message is transmitted separately for each recipient. While a single message can be delivered to several recipients by using several RCPT TO commands, the MAIL FROM command can be used only once for each message. Finally, the delivery of messages can be delayed due to graylisting if the MAIL FROM address includes a value which is unique to each message. Unique MAIL FROM addresses allow the mailing list software to identify which particular message could not be delivered to a particular recipient.

▼ Backscatter

Given how easy it is to spoof the sender address, it's sometimes better not to send a bounce message. Otherwise, the owner of the forged address might receive a large number of unsolicited bounce messages. Such collateral spam is called backscatter. In order to distinguish legitimate bounce messages from misdirected ones, the outgoing mail server can authenticate the bounce address by extending it with a hash-based message authentication code (HMAC). This allows the incoming mail server to reject bounce messages which are addressed to a non-authenticated address. The best-known proposal for how to do this is called bounce address tag validation (BATV). It is specified in this draft. In order to prevent the authenticated bounce address from being abused, the HMAC is calculated over the original MAIL FROM address and a timestamp of when the authenticated address expires. The timestamp and some part of the HMAC are prepended to the original MAIL FROM address to form the authenticated bounce address.

Password-based authentication mechanisms

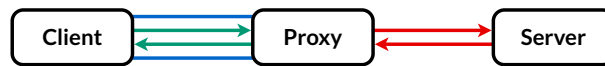
The following boxes focus on password-based authentication mechanisms, which allow users to authenticate themselves to servers with only their username and password. Due to the nature of the topic, some of the later information boxes are fairly advanced. If you're not interested in cryptography, you may want to skip them.

▼ Dangerous reliance on TLS

As we've seen above, your password is usually sent to the outgoing mail server every time you submit a message. Many people think that this is no problem because the password is transmitted over a secure channel. The goal of this box is to convince you that this attitude is naive and dangerous. In the remaining boxes of this subsection, I will explain how we could do much better.

One of the most important principles in information security is defense in depth: Critical systems should have several layers of protections so that when one layer fails, another can stop the threat. In risk analysis, this is sometimes referred to as the Swiss cheese model. There are three different ways in which Transport Layer Security (TLS) can fail to protect sensitive information:

- **Proxy server:** TLS connections are often terminated at a so-called proxy server, which acts as an intermediary between the client and the actual server. While such proxies are operated by the same company as the actual server, the communication between the client and the server is no longer protected between the proxy and the actual server in the company's private network. Running a proxy which appears to the client as the actual server is useful for load balancing and for accelerating the cryptographic operations with special hardware. On the downside, an employee or an attacker who compromised the company's network has potentially access to the transmitted information, which is no longer protected by TLS.



While the communication between the client and the proxy is protected (indicated by the blue lines), the communication between the proxy and the server is exposed in the company's private network.

- **Wrong server:** The user's mail client might be misconfigured and connect to a server controlled by the attacker. Instead of communicating with the email service provider, the client communicates with the attacker. In order to avoid detection by not raising any suspicion, the attacker may want to connect to the legitimate server themselves and relay all messages in both directions. Both the client and the legitimate server have the impression that they communicate with the other party over a secure channel when in fact the attacker can read and modify the exchanged messages at will. But why would the mail client be misconfigured? On the one hand, the user might fall for a social engineering attack. While phishing is much easier when it comes to websites because the user just has to click on a malicious link, it's also possible in the case of mail clients: The user just has to follow malicious instructions. On the other hand, the mail client might be attacked during autoconfiguration. Possible attack vectors are spoofed DNS entries if the client doesn't require DNSSEC or a compromised configuration database. Even if the client checks the domain name with some heuristic, the heuristic might be vulnerable to similar attacks, especially in the case of custom domains.



The client is misconfigured and connects directly to the attacker, who forwards all communication without raising any suspicion.

- **Compromised certification authority or compromised server key:** While the public-key infrastructure worked as intended in the previous example (the malicious server had a legitimate certificate for its domain name), the current infrastructure is far from perfect. Its biggest design flaw is that any vendor-approved certification authority (CA) can issue certificates for any domain by default. There have been several attacks on the integrity of TLS, ranging from compromised and misbehaving certification authorities to surveillance programs and search warrants for private keys. In other words, TLS isn't guaranteed to be secure, but it's still much better than having no protection at all, of course. An illegitimately issued certificate allows the attacker to intercept the communication between the client and the server. As long as the client considers the certificate to be valid, it will accept the certificate and start communicating with what it believes to be the intended server. There have been several efforts to prevent certification authorities from issuing certificates without the consent of the domain owner, such as HTTP Public-Key Pinning (HPKP) and Certificate Transparency (CT). We will discuss another approach in the last chapter of this article. It's important that these efforts don't remain limited to the web but that mail servers begin to require more secure certificates as well. As I will explain to you in the following boxes, we don't even need better certificates to protect the communication between mail clients and mail servers, simply using better authentication mechanisms would be enough. And unlike the efforts at the security layer, which prevent only attacks with maliciously issued certificates, better authentication mechanisms also prevent attacks with compromised server keys, where an attacker has gained access to the server's private key. When I say that an organization or key is compromised, I just mean that its behavior or use is different from what it should be according to standards and agreements. Whether the integrity was lost due to an attack or due to deliberate actions by the owner doesn't matter. For a security analysis, it's also irrelevant whether the perpetrator acts in good faith or in bad faith. This article is not about the ethics of information security and government backdoors.

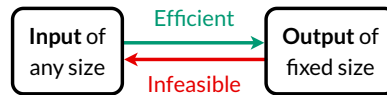


The malicious server (in red) has the same name as the legitimate server (in green). An attacker can impersonate the server by getting a certificate issued for their public key or by gaining access to the private key of the server and using the original server certificate.

In these scenarios, the attacker is a so-called man-in-the-middle (MITM) in the conversation between the client and the server.

▼ Cryptographic hash functions

Before we can discuss better authentication mechanisms, we have to cover cryptographic hash functions first. A cryptographic hash function is an algorithm, which maps inputs of arbitrary size to outputs of fixed size deterministically and irreversibly:



Cryptographic hash functions are efficient to compute and infeasible to invert.
For the same hash function, the same input always maps to the same output.
The output is also called image and the corresponding input its preimage.

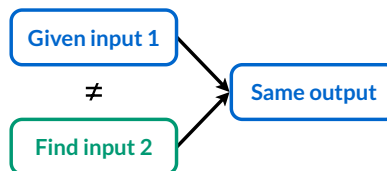
The output of a hash function is called the hash of the input. As a verb, hashing refers to applying the hash function to an input. More formally, cryptographic hash functions have to fulfill the following properties. A function which maps arbitrary inputs to fixed-sized outputs without fulfilling these properties is just a hash function. Unless noted otherwise, I always mean the former.

- **Preimage resistance** (also known as one-way function): It's infeasible to find an input which hashes to a given output.



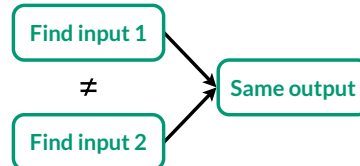
In these graphics, the given values are displayed in blue and the values to find in green.

- **Second preimage resistance**: It's infeasible to find a different input which hashes to the same output as a given input.



Knowing one input may not be useful to find another input which hashes to the same output.

- **Collision resistance**: It's infeasible to find two different inputs which hash to the same output, resulting in a collision.



Due to the birthday paradox and attack, this is a stronger requirement than the previous one.

Since hash functions map an infinite number of inputs to a finite number of outputs, they have to produce an infinite number of collisions. The point of cryptographic hash functions is not that they don't have any collisions, it just has to be infeasible to find them. In practice, cryptographic hash functions should also satisfy the avalanche criterion: A small change in the input changes the output completely and unpredictably. A cryptographic hash function is said to be broken if a preimage or a collision can be found more efficiently than with a brute-force search or if the size of the output is so small that a brute-force search becomes feasible with modern computers.

Regarding notation, I will use : to assign a value on the right to a variable on the left in the following boxes. For example, a hash function is then written as Output: hash(Input). Sometimes, several values need to be combined into one before hashing. I'll use + to concatenate several values in a secure way: Output: hash(Input1 + Input2). When implementing this, you can use a special character which may not occur in any of the values as a delimiter so that hash("a" + "bc") ≠ hash("ab" + "c"). The null character is used for this purpose in the case of PLAIN authentication.

The term "hash" is most likely borrowed from the kitchen, where it means to chop and mix ingredients when preparing food.

▼ Secure Hash Algorithms (SHA)

The National Institute of Standards and Technology (NIST) standardizes cryptographic hash functions under the name Secure Hash Algorithms (SHA). NIST published several Secure Hash Algorithms so far: SHA-1 in 1995, SHA-2 in 2001, and SHA-3 in 2015. The most commonly used cryptographic hash function is SHA-256. You can try it and other hash functions with this tool:

Input: Algorithm: Uppercase: ☐


```
$ printf '%s' 'Hello' | openssl sha256
185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```

The three digits at the end of some algorithm names indicate the size of the output in bits. SHA-256, for example, hashes inputs of arbitrary size to 256 bits. SHA-224, SHA-256, SHA-384, and SHA-512 belong to the SHA-2 family of hash functions. Collisions have been found for MD5 and SHA-1, which hash to 128 and 160 bits respectively. These algorithms should no longer be used.

▼ Salts against pooled brute-force attacks

The one-way property of cryptographic hash functions doesn't prevent an attacker from generating and testing possible inputs. If the set of possible (or likely) inputs is small enough, it can be searched exhaustively to find the preimage of a given hash. If you try to find the preimage of many hashes, you have to compute the hash of possible inputs only once. If you want to find further preimages in the future, you can store the computed input-output pairs in a reverse lookup table. Instead of trying all possible inputs again, which can take a long time, you simply look up the hash to crack in the output column of your precomputed table. Since computers have limited memory, this approach works only for a limited number of inputs. You can enumerate all possible inputs up to a certain length or choose them from a dictionary. You can reduce the amount of required memory by accepting longer lookup times. This is known as a space-time tradeoff and is achieved with so-called rainbow tables.

Searching for the preimage of many hashes at once can be prevented by adding a random value to each input and storing this value together with the output. While an attacker can still generate and test possible inputs, they have to spend the required effort on each hash separately. The additional input value is called salt. In order to have the intended effect, the salt has to be chosen at random for each input and should be as long as the output size of the used hash function.

Why is the random value called salt? No one really knows. When cooking, salt is something you add to your ingredients before mixing them. With enough salt, you can make food unenjoyable. Salting the earth is also a historic practice to make land less hospitable for your enemy. We also say to take something with a grain of salt. Whatever the origin may be, the term fits well.

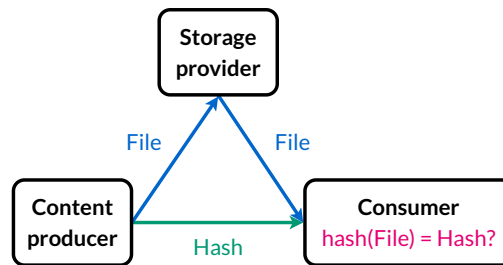
▼ Nonces against replay attacks

When designing a cryptographic protocol, you not only want to ensure that an attacker cannot produce certain messages, you also want to ensure that an attacker cannot record such messages and reuse them at a later point against one of the legitimate parties. Such replay attacks are prevented by including a number which may be used only once, which is abbreviated to nonce. The replay of messages needs to be prevented both within a session and across sessions. The former is typically accomplished with a counter: A message is accepted if its counter is higher than the previous counter. The latter is usually achieved by using a random number for the duration of a session so that no information has to be persisted across sessions. Instead of including a session-specific value in each message, choosing some of the cryptographic keys randomly for each session has the same effect. When the uniqueness is incorporated into temporary keys, we no longer speak of nonces but rather of ephemeral keys. Unlike salts, which stay the same throughout the lifetime of a hash and need to be stored, nonces and ephemeral keys can be thrown away after use. Besides preventing replay attacks, mixing some uniqueness into every message has the desirable side effect of preventing an attacker from learning when the same underlying value is sent again by one of the parties.

▼ Applications of cryptographic hash functions

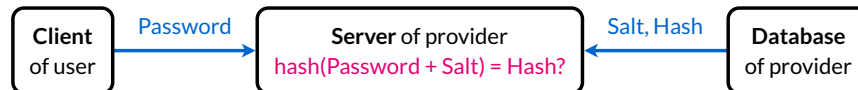
Before moving on to authentication mechanisms, I first want to mention some applications of cryptographic hash functions:

- **Data integrity:** Due to their collision resistance, cryptographic hash functions produce a unique fingerprint of the data which was fed into them. In other words, the hash of a file uniquely identifies the file. As long as you get the short hash from a trusted source, the large file can be downloaded from an untrusted source because you can detect potentially malicious changes to the file by computing the hash of the file and comparing it with the trusted hash. You can compute the hash of a file with `openssl sha256 /path/to/file`. Eliminating trust in the storage provider is really useful for content delivery networks (CDN), which you might have encountered as mirror sites or as subresource integrity (SRI) on the Web. This fingerprint property is also used for digital signatures, where you sign the hash of a message rather than the message.



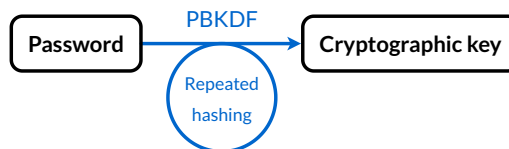
As long as you get the hash from a trusted source, the delivery of the file can be outsourced to an untrusted third party.

- **Password protection:** In order to verify whether a user provided the correct password, a server doesn't have to store the password of the user. The server can simply store a salted hash of the password and then check whether the user provided the same password as before by computing and comparing its hash. The advantage of this approach is that an attacker who compromised the database cannot log in as the user as they don't know the preimage of the salted hash.



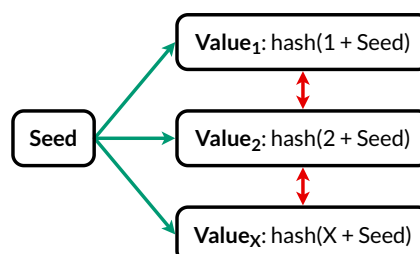
A server can reduce the damage of a leaked database by storing individually salted hashes instead of passwords.

- **Key derivation:** Cryptographic hash functions are designed to run as fast as possible. While good performance is desirable for many applications, it's not desirable when hashing passwords. Even if the hash of a password is salted, an attacker can still perform a brute-force attack to find an input which hashes to the given output with the given salt. In order to make such attacks costlier, passwords are often hashed thousands of times instead of just once. Repeated hashing means that you take the output of one round as the input to the next round. This also makes the computation costlier for the legitimate parties but unlike an attacker, they have to compute the derivation only once per session. Making a weak key more secure against brute-force attacks by increasing the cost is called key stretching. One algorithm for doing so is the Password-Based Key Derivation Function 2 (PBKDF2), which is specified in RFC 8018. Additionally, cryptographic keys typically have a desired length, which is another reason for using a key derivation function (KDF).



By hashing an input repeatedly, you can turn an efficient hash function into an inefficient one.

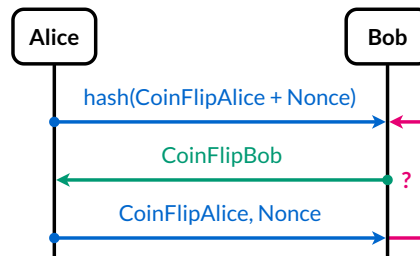
- **Independent values:** Another use case of cryptographic hash functions is to generate a sequence of unrelatable values from a single source value. Such a source value is called a seed because a tree of values can grow from it. The seed is then hashed with a counter or a timestamp. As long as the seed remains secret, others cannot compute the next value from the previous one and vice versa. Hash functions are used for this purpose in contact tracing apps, cryptocurrency wallets, and one-time passwords (OTP). If a hash function fulfills the strict avalanche criterion, it can even be used as a pseudo-random number generator (PRNG) or as a block cipher for encryption. For all these use cases, the seed has to be chosen randomly, which means it has to have enough entropy. If you don't like password managers, you can use hash functions to generate site-specific passwords as `SitePassword: hash(LoginDomain + MasterPassword)`. Unless you know exactly what you're doing, I advise you not to use this technique as there are many pitfalls, such as leaving your password in the command history or accidentally including newline characters, but it's certainly a neat idea. (The order of LoginDomain and MasterPassword is important as you might be vulnerable to a length-extension attack otherwise, see below.)



Values can easily be derived from a seed (in green) but they cannot be related to one another (in red).

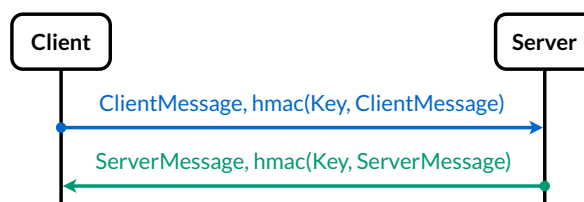
- **Commitment schemes:** A commitment scheme allows you to commit yourself to a value while keeping the value secret until you reveal it later. You can think of it as giving a locked box to a recipient while providing the key to open the box only later. A commitment scheme has to be both binding and hiding: The committer may not be able to change the committed value and the recipient may not be able to figure out the committed value. In order to understand why this is useful, let's look at an example from Wikipedia. Suppose Alice and Bob need

to resolve a dispute over the Internet. If they were at the same place, they could simply flip a coin. Since they are remote, one would have to trust the other to report the flip correctly. As neither of them is willing to trust the other, they come up with the following procedure. Alice flips a coin and hashes the outcome with a random nonce. She then sends the output to Bob, who replies with the outcome of his own coin flip. Finally, Alice reveals her commitment by sending her coin flip and nonce to Bob. By verifying whether the flip and the nonce hash to the value he received earlier, Bob can detect if Alice attempted to cheat. Alice and Bob agreed that if their coin flips are the same, then Alice wins. If not, Bob wins. If the hash function is secure, neither of them can skew the result in their favor.



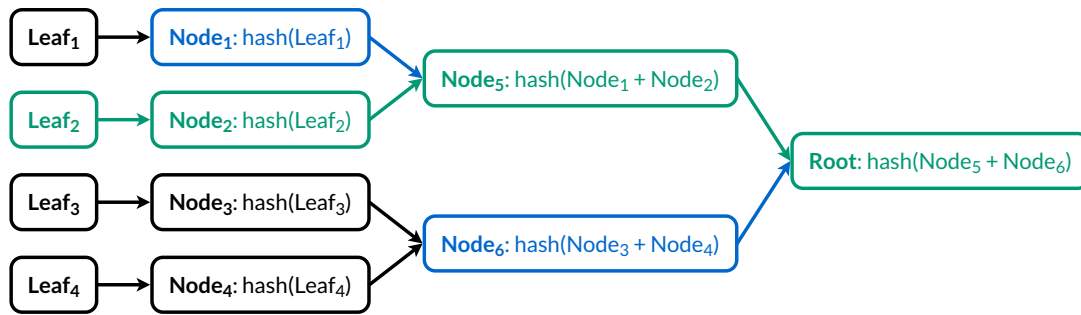
If $\text{CoinFlipAlice} = \text{CoinFlipBob}$, Alice wins. If $\text{CoinFlipAlice} \neq \text{CoinFlipBob}$, Bob wins.

- **Message authentication:** How can two parties be sure that no one tampered with their communication? They can achieve this by extending each message with a value which depends on the message and which only they can generate. Such a value is called a message authentication code (MAC). If an attacker modifies a message, the original MAC no longer matches the message and the attacker cannot fix this because they cannot generate a valid MAC. Both parties compute the MAC for each message they receive and reject all messages for which the transmitted MAC is different from the computed MAC. One way of implementing message authentication codes is to hash the message together with a value which is known only to the legitimate parties. This value is a shared secret and it is used as a cryptographic key. For example, the MAC could be computed as $\text{hash}(\text{Key} + \text{Message})$. Unfortunately, this isn't secure when used with any of the hash functions listed above as they are all vulnerable to length-extension attacks. The problem is that these algorithms leak their internal state as the result, so an attacker can simply continue where the legitimate party left off without having to know the shared key. This means that, given a Message and the corresponding MAC, an attacker can generate a valid MAC for the message $\text{Message} + \text{MaliciousAddition}$. While swapping the key and the message solves this problem, $\text{hash}(\text{Message} + \text{Key})$ makes the MAC immediately vulnerable as soon as the hash function becomes vulnerable to collision attacks. In order to avoid such issues, cryptographers came up with the Hash-based Message Authentication Code (HMAC) in 1996, which is defined as follows: $\text{hmac}(\text{Key}, \text{Message}) = \text{hash}([\text{Key}' \circ \text{OuterPadding}] + \text{hash}([\text{Key}' \circ \text{InnerPadding}] + \text{Message}))$, where the \circ denotes the bitwise exclusive-or operation and the square brackets are used only to make the parenthesis matching easier. The paddings are the same for everyone and their purpose is to make the key in the inner hash different from the key in the outer hash. If the key is longer than the block size of the used hash function, it needs to be hashed: $\text{Key}' = \text{hash}(\text{Key})$. Not always hashing the key leads to trivial collisions, which should have been avoided when specifying the algorithm. As long as you understand what HMAC is good for, the details don't matter here. The SHA-3 algorithms aren't susceptible to length-extension attacks and my understanding is that the much simpler $\text{hash}(\text{Key} + \text{Message})$ construction works as intended with them. What is important to note is that hash-based message authentication codes are symmetric: Whoever can verify them can also generate them. Unlike digital signatures, message authentication codes allow a party to repudiate messages which it authenticated because the other party could have generated the corresponding MAC as well.



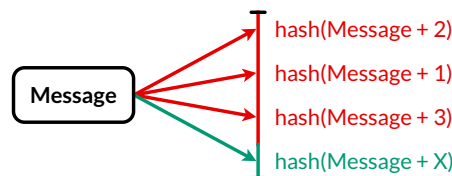
A message authentication code is appended to each message.

- **Proof of inclusion:** When collaborating online, it's sometimes useful to be able to prove to others that a record has been incorporated into the current state of a system without having to share or even disclose all the other records. This can be accomplished by repeatedly hashing two hashes into one until you're left with a single hash which captures the state of the whole system. The resulting structure is called a Merkle tree. Records cannot be added to, removed from, or modified in this structure without affecting the so-called root of the tree. If someone accepts that a specific root represents the state of the system, you can prove to this person that a particular record is included in this state by revealing the hash of the branches with which this record has to be hashed in order to arrive at this root. This method is interesting for two reasons: The proof grows logarithmically with the number of records, which makes it scale very well, and the other records have to be neither revealed nor transmitted for the verification to succeed. Such proofs of inclusion are used in Bitcoin for Simplified Payment Verification (SPV), in decentralized timestamping for document aggregation, and in Certificate Transparency for auditing.



In order to verify that the green leaf is included in the root, a verifier needs to know only the hashes and positions of the blue nodes.

- **Proof of work:** In publicly accessible systems, you want to discourage participants from using a shared and limited resource beyond their fair share. One way of doing so is by imposing a cost on using the resource, which deters anyone who doesn't value the resource higher than its associated cost. The resource owner can either charge a fee for using the resource or require its users to waste a limited resource of their own. While the former approach is less wasteful, the latter approach doesn't require a global infrastructure for micropayments. For example, your mailbox is a publicly accessible system and your time is a limited resource. What if you could require every unknown sender to waste one minute of computing power before they can deliver an email to your inbox? This would prevent spammers from sending millions of emails a day – or at least make this antisocial behavior much costlier. It turns out that there's a simple way to achieve this: You could require that the hash of incoming messages falls into a tiny range. Since one cannot influence the output of a hash function, senders have to keep appending different nonces to their message until its hash finally falls into the desired range. As long as the hash function isn't broken, there's no better way than to keep trying until you're lucky. It's like trying to hit the bull's eye on a target when you have zero control over the trajectory of your darts. While finding an appropriate nonce requires many computations, the recipient has to hash the message just once in order to verify whether the required work has been done. The average difficulty of the problem can be adjusted by making the target range bigger or smaller. This technique was invented in 1992 as a digital postage stamp but saw widespread usage only with the rise of cryptocurrency mining.



Finding a nonce which makes the hash of a message fall into a certain range requires many attempts.

▼ Exclusive-or operation for perfect encryption

Exclusive or is a binary truth function, which means that it combines two inputs into a single output, where all values are either true or false. Exclusive or returns true if one of the inputs is true but not both. Instead of true and false, we will use the symbols 1 and 0. The operator is often written as a plus in a circle because it corresponds to binary addition without the carry. Functions which map a finite combination of inputs to some output can be specified simply by listing all possible mappings in a table:

A	⊕	B	=	C
0		0		0
0		1		1
1		0		1
1		1		0

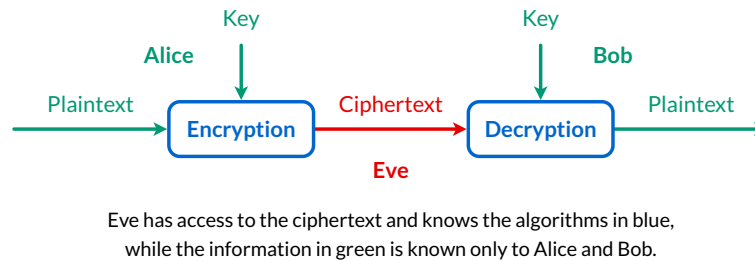
The truth table of the exclusive-or operation. The output is 1 if and only if the two inputs are unequal.

Since the above table is exhaustive, you can convince yourself that the following properties hold simply by studying all cases:

- **Commutativity:** The order of the inputs doesn't matter: $A \oplus B = B \oplus A$.
- **Reversibility:** Applying \oplus to the output and one of the inputs gives you the other input: $C \oplus B = A$ and $C \oplus A = B$.
- **Entropy-preservation:** If one of the inputs has a 50% probability of being 0 and a 50% probability of being 1, then the output also has a 50% probability of being 0 and a 50% probability of being 1, independent of whether the other input is 0 or 1: $A \oplus (50\%: 0, 50\%: 1) = (50\%: 0, 50\%: 1)$. In other words, if one of the inputs is truly random, then so is the output. In information theory, the amount of information in a random variable is called entropy. As long as two random variables are statistically independent from one another, combining them with exclusive or can only increase the entropy but never decrease it.

Instead of applying a binary function on binary inputs to two single bits, we can also apply it to two equally long strings of bits by combining the bits at each position separately. Every such function has a bitwise equivalent, which is usually denoted with the same or a similar symbol. For example, $0011 \oplus 0101 = 0110$, which corresponds to the columns in the above table.

Encryption enables two parties to transfer confidential information over an insecure channel. In examples, the parties are typically called Alice and Bob, while the eavesdropper, who tries to listen in on their conversation, is usually called Eve. The unencrypted message is called plaintext, the encrypted message is called ciphertext. In symmetric-key cryptography, Alice and Bob use the same piece of information, which is known as a cryptographic key, to encrypt and decrypt the message. According to Kerckhoffs's principle, the two algorithms should be designed such that the encryption scheme is secure even if the enemy knows them. It's considered bad practice to achieve security through obscurity. The following graphic depicts all these terms:



The bitwise exclusive-or operation can be used to construct an encryption scheme, which is known as the one-time pad. In order to encrypt a message, Alice computes $\text{Ciphertext} = \text{Plaintext} \oplus \text{Key}$. Bob can then decrypt the message by computing $\text{Plaintext} = \text{Ciphertext} \oplus \text{Key}$ thanks to the reversibility property of the exclusive-or operation. According to the entropy-preservation property, if the key is completely random, then so is the ciphertext. To put it differently: If every bit of the key has a 50% probability of being 0 and a 50% probability of being 1, the same is true for every bit of the ciphertext, regardless of what the plaintext looks like. Since the ciphertext contains no information at all about the plaintext, the one-time pad is information-theoretically secure, which means that even an adversary with infinite computing power cannot break the encryption scheme. Trying all possible keys doesn't work because for every possible plaintext there's a key ($\text{Key} = \text{Plaintext} \oplus \text{Ciphertext}$) which produces the observed ciphertext. While this encryption scheme is perfectly secure, it's rarely used in practice because the key has to be at least as long as the plaintext and each key may be used to encrypt only a single message; hence the name one-time pad. Practical encryption schemes derive an infinite sequence of key material from a finite value, sacrificing perfect security by doing so. This makes the distribution of keys much easier, either by sharing them in advance or by deriving them when needed.

The one-time pad encryption scheme is highly malleable. An attacker can truncate the message at an arbitrary position and flip bits in the ciphertext to cause a bit flip at the same position in the plaintext, which allows the attacker to replace all parts of the plaintext which are known to them. Such attacks can be prevented by appending a checksum to the plaintext before encrypting it and requiring the recipient to validate the checksum after decryption. We'll revisit this in the last section of this article.

▼ Desirable properties of authentication mechanisms

Now that we've covered the cryptographic concepts that we will need (namely hash functions, salts, nonces, key derivation functions, message authentication codes, and exclusive or), we can turn our attention to password-based authentication mechanisms. What makes them interesting is that we want to arrive at strong security from relatively weak passwords.

Unfortunately, I couldn't find any good literature on desirable properties of password-based authentication mechanisms, which is why I made up the following criteria myself. Since this isn't my area of expertise, let me know if I missed an important aspect. (Section 5 of RFC 7616 is the best source that I could find, covering security considerations of Digest Access Authentication.)

An ideal password-based authentication mechanism is resistant to:

1. **Database compromise:** An attacker who compromised the server's authentication database cannot impersonate its users. For this reason, passwords should never be stored in plaintext. Given that an attacker who compromised the database no longer has to interact with the server, there is no limit in the number of passwords they can try every second. In order to increase resistance against such offline brute-force attacks, passwords should be individually salted and repeatedly hashed. While authentication mechanisms usually don't dictate how servers have to store the information required to authenticate their users, their design can prevent the service provider from applying certain techniques such as salting and stretching.
2. **Replay attacks:** An attacker who intercepts the communication between the client and the server cannot impersonate the user in self-initiated sessions. This is accomplished by making the transmitted authentication information valid only in the current session, which limits the harm that a man-in-the-middle can cause to the current session. This is especially valuable if the client can demand only a single action per session from the server, such as submitting a single message to the outgoing mail server per connection. Unfortunately, this isn't the case for any of the protocols discussed in this article. By preventing delayed attacks, resistance to replay attacks is still a desirable property because it makes attacks much easier to localize.

3. **Pooled brute-force attacks:** An attacker who compromised the communication channel of several users cannot generate and test password candidates for several users at once.
4. **Individual brute-force attacks:** An attacker who compromised a communication channel encounters only stretched derivations from the password, which makes brute-force attacks costlier.
5. **Denial-of-service attacks:** An attacker cannot launch a computational [denial-of-service attack](#) against clients.
6. **Server impersonation:** An attacker cannot impersonate a server towards a client without relaying the authentication messages to the actual server. When combined with measures against man-in-the-middle attacks, this prevents sending sensitive information to an attacker who just fakes that the authentication was successful without knowing whether this is the case. For example, a client shouldn't submit an email to a server which cannot verify whether the password was correct.
7. **Wrong server:** The client detects when it's connected to the wrong server (see the box on our [dangerous reliance on TLS](#)).
8. **Compromised certification authority:** The client detects when the used certificate doesn't belong to the actual server.
9. **Compromised server key:** The client detects when the server is impersonated even if the same certificate is being used.
10. **Comparison attacks:** A compromised server cannot learn whether two different accounts are protected with the same password, neither when creating an account nor during ordinary authentication. Such knowledge can be used to infer that the accounts belong to the same person – or to contact and bribe one user to compromise the password of the other user.
11. **Wrong server after database compromise:** There is no risk in connecting to the wrong server even if the server's database has been compromised. This property is desirable because the database compromise might remain undetected. And even if the data breach is detected, many users are likely too lazy to change their passwords. The wrong server might also just be another server where a user uses the same password. Reusing the same password should be secure even if you don't trust all service providers. (The "should" refers to an ideal authentication mechanism, which is implemented with static code on the client. Don't reuse the same password on different websites! I have a separate box about [authentication on the Web](#).)
12. **User impersonation after server compromise:** An attacker who compromised the server cannot impersonate its users. This means that even if the server was compromised temporarily, users don't have to change their password. Additionally, this property guarantees the user that the server is resistant to a database compromise. The database could even be public.

The goal of defense in depth is to limit the potential harm as much as possible. Given that you can reset the password of many of your online accounts through your email account, you don't want to send one of your most valuable passwords directly to a potential attacker when checking your inbox. Let's look on how the [three authentication mechanisms](#) perform in this regard:

Resistant to	PLAIN	CRAM	SCRAM
Database compromise	✗	✗	✓
Replay attacks	✗	✓	✓
Pooled brute-force attacks	✗	✗	✓
Individual brute-force attacks	✗	✗	⚠
Denial-of-service attacks	✓	✓	⚠
Server impersonation	✗	✗	✓
Wrong server	✗	✗	⚠
Compromised certification authority	✗	✗	⚠
Compromised server keys	✗	✗	⚠
Comparison attacks	✗	✗	⚠
Wrong server after database compromise	✗	✗	⚠
User impersonation after server compromise	✗	✗	✗

A comparison between password-based authentication mechanisms.

✓ means that the authentication mechanism is resistant to the attack.

⚠ means that the resistance depends on choices made by programmers.

✗ means that the authentication mechanism is vulnerable to the attack.

Unfortunately, only the [PLAIN](#) authentication mechanism is widely deployed on mail servers. Before we discuss how [CRAM](#) and [SCRAM](#) do or don't fulfill the above properties, let me mention some aspects which are beyond the scope of this analysis:

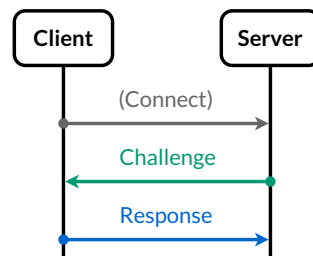
1. **Bugs in implementation:** Even if an authentication mechanism is resistant to an attack in theory, it can be vulnerable to it in practice because of [software bugs](#). All you can do is to [actively look for them](#), [encourage their disclosure](#), and fix them soon.
2. **Account theft:** Authentication mechanisms usually don't specify how users set and change their password. An attacker who intercepts the daily communication between a client and a server shouldn't be able to [change the user's password](#), thereby stealing their account.

Since changing the password is even beyond the scope of most protocols, there isn't much to say about this here other than that it's a problem of authorization rather than a problem of authentication. According to the principle of least privilege, the credentials required for changing the password are ideally different from the ones required for accessing the system. OAuth achieves this with restricted scopes. Some email service providers (e.g. Apple and Google) allow users to generate app-specific passwords, which can be revoked individually and which aren't enough to change the user's password. Given that PLAIN is the dominant authentication mechanism, app-specific passwords are highly desirable.

3. **Downgrade attacks:** If a server supports several authentication mechanisms, a man-in-the-middle can remove the stronger ones so that the client is forced to continue with the weakest one. We discussed measures against downgrade attacks in the context of backward compatibility earlier. New services can support only the strongest authentication mechanism, which eliminates this problem as well. The weakness here lies not in individual mechanisms but rather in how they are deployed.
4. **Online attacks:** If the attacker has to interact repeatedly with one of the legitimate parties, we speak of an online attack. Since users should be able to authenticate themselves from a different network, an attacker can do the same interaction with one guessed password at a time. Due to the nature of authentication mechanisms, online attacks are always possible. However, service providers can make them more difficult by limiting the rate at which new passwords can be tried and by informing the user about failed attempts. If not implemented carefully, legitimate users can also be affected by rate limiting.
5. **Server compromise:** As long as the server is compromised, there's nothing left to protect by an authentication mechanism.
6. **Client compromise:** An authentication mechanism cannot prevent users from entering their password into a compromised client. The harm can be limited only by using app-specific passwords or OAuth (see the second point about account theft).
7. **Compromised certification authority after database compromise or compromised server keys after database compromise:** What I write here will make more sense once you've read the box on SCRAM. An attacker who compromised the database succeeds in the mutual authentication towards the client. Since the relay of messages to the actual server is therefore no longer necessary, channel binding can no longer prevent these two variants of the man-in-the-middle attack.

▼ Challenge-Response Authentication Mechanism (CRAM)

CRAM is a very simple authentication mechanism, in which the client has to respond to the challenge received from the server:



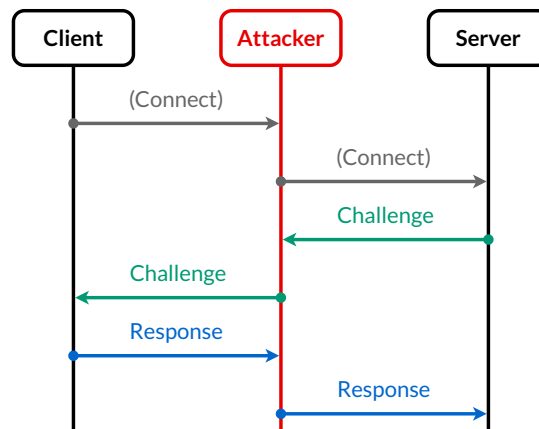
How challenge-response authentication works.

CRAM is specified in RFC 2195 and draft-ietf-sasl-crammd5. Unlike what some documentation suggests, CRAM has nothing to do with encryption. The client computes the response as $\text{Response} = \text{hmac}(\text{Password}, \text{Challenge})$, where the challenge was chosen randomly by the server. The HMAC could be instantiated with any hash function but the standard uses MD5, which is why the full name of the mechanism is CRAM-MD5. Let's evaluate which of the above properties are fulfilled by CRAM-MD5:

1. **Database compromise:** In order to verify the response, the server needs to be able to perform the same computation as the client. Since the password is directly used as an input to the HMAC, the server has to store the password rather than its salted hash. In this regard, CRAM is worse than PLAIN, where only a derivation of the password needs to be stored. Both the RFC and the draft say that the security can be marginally improved by storing the state of the hash function after feeding in the password instead of the password itself. This is putting the length-extension vulnerability of many hash functions such as MD5 to supposedly good use. However, this doesn't help at all because if the server can continue from the intermediary state to determine the response, then so can the attacker who compromised the database and tries to impersonate users.
2. **Replay attacks:** As long as the server never issues the same challenge twice, the response from the client is valid only in the current session. If an attacker replays an old response to a new challenge, the server rejects the received value as invalid.
3. **Pooled brute-force attacks:** As a man-in-the-middle, the attacker can send the same challenge to several clients. Therefore, the attacker can test password candidates for several users at once.
4. **Individual brute-force attacks:** An authentication mechanism which isn't resistant to pooled brute-force attacks is also not resistant to individual brute-force attacks.
5. **Denial-of-service attacks:** Since the number of hashes that a client has to compute per authentication is fixed, denial-of-service attacks against the client aren't possible (as long as the size of the challenge is limited by the protocol).

6. **Server impersonation:** Since the client doesn't authenticate the server, an attacker can impersonate the server in one of the above-mentioned ways and fake the success of the user authentication.
7. **Wrong server:** The client cannot detect when it's connected to the wrong server.
8. **Compromised certification authority:** The client cannot detect when the used certificate doesn't belong to the real server.
9. **Compromised server key:** The client cannot detect when the server is impersonated if the same certificate is being used.
10. **Comparison attacks:** Since the server stores the passwords, it can easily determine if two accounts use the same password.
11. **Wrong server after database compromise:** If the server's authentication database has been compromised, users have to change their password wherever they're using it.
12. **User impersonation after server compromise:** An attacker who has compromised the server can impersonate its users.

Before we move on to SCRAM, I wanted to visualize how a man-in-the-middle can relay all messages between the two parties:



A man-in-the-middle attack on a challenge-response authentication mechanism.

▼ Salted Challenge-Response Authentication Mechanism (SCRAM)

Looking at its name, SCRAM seems to be just a salted version of CRAM. This is misleading, however, as SCRAM is much more than that. SCRAM is specified in RFC 5802 and improves on CRAM with the following, now mostly familiar techniques:

- **Key derivation:** Instead of using the password directly, SCRAM uses PBKDF2 to derive a cryptographic key. By salting the password and hashing it thousands of times, a brute-force search for the password given the key becomes very costly.
- **Message authentication:** The derived key is used to authenticate a message from the client to the server and a message from the server to the client with an HMAC. The server can authenticate its message only if it knows the derived key. We thus have mutual authentication: The server is certain that the user is who they claim to be and the client is certain that the message came from the right server.
- **Exclusive-or encryption:** The problem is that the server shouldn't store this key. Otherwise, anyone who compromised its database can impersonate the user by authenticating the appropriate message with the stolen key. This can be solved by storing a hash of the derived key instead. Note that the derived key doesn't need to be salted and stretched here because the best way to find the preimage of the hashed key is to guess the low-entropy password, which is itself already salted and stretched to arrive at the derived key. The client then uses the hashed key to authenticate its message. So far we have only moved the problem, though, because the hashed key now has the same role as the derived key before. The trick is that the client proves to the server that it knows the preimage of the stored key by encrypting the preimage with the HMAC. Since only the legitimate parties can compute the HMAC, the server can decrypt the preimage but the attacker cannot. If this is confusing, then re-read this paragraph after you've seen the protocol flow below.
- **Optional channel binding:** The authenticated message includes everything which the client and the server have to agree on. One useful thing to agree on is that they are connected to the same secure channel. Binding the channel on the application layer to the channel on the security layer prevents man-in-the-middle attacks. Channel binding is optional in SCRAM. There are different ways to bind the inner channel to the outer channel with different tradeoffs. We'll cover them in the next box.



Mutual authentication guarantees only that the inner channel (in green) reaches the counterparty.
Channel binding can be used to ensure that the outer channel (in blue) isn't interrupted by an attacker.

What follows is a simplified version of the SCRAM protocol. I believe it has the same properties as the official protocol and I'm not aware of any vulnerabilities. However, be aware that my simplifications haven't been reviewed. The SCRAM standard might do things differently for good reasons, which I just haven't thought of. For the sake of compatibility and security, implement the official protocol! I simplified the

protocol only to make it easier to understand. The biggest differences are that I don't separate the "server key" from the "client key" and that I removed the redundancy in the transmitted and thus authenticated messages. Reducing the number of variables allows me to use less confusing names for them. I didn't just simplify SCRAM, though, I also provide suggestions for improving SCRAM in my analysis below. Let's have a look now at how Simplified-SCRAM works.

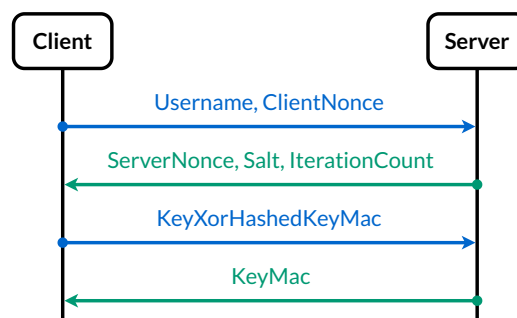
As with every password-based authentication mechanism, the user's credentials are Username and Password. We also have:

- Salt and IterationCount: The values to derive the Key from the Password. The RFC doesn't specify who chooses the Salt.
- ClientNonce and ServerNonce: Values chosen at random for each session. The former by the client, the latter by the server.
- ChannelBinding: A string which identifies the TLS channel over which the messages are sent. See the [next box](#) for options.

The client and the server compute the following values based on the above values:

- Key: $\text{pbkdf2}(\text{Password}, \text{Salt}, \text{IterationCount})$
- HashedKey: $\text{hash}(\text{Key})$
- Message: $\text{Username} + \text{ClientNonce} + \text{ServerNonce} + \text{ChannelBinding}$
- HashedKeyMac: $\text{hmac}(\text{HashedKey}, \text{Message})$
- KeyXorHashedKeyMac: $\text{Key} \oplus \text{HashedKeyMac}$
- KeyMac: $\text{hmac}(\text{Key}, \text{Message})$

For each user, the server stores Username, Salt, IterationCount, and HashedKey. The following messages are exchanged:



The sequence diagram of Simplified-SCRAM.

Since a user has to be able to authenticate themselves on a new client with just their Username and Password, the server has to store the Salt and the IterationCount and provide it to the client on request. Since the user is not yet authenticated at this stage, anyone can request the Salt and the IterationCount of any user. (The IterationCount determines how many times the salted password is hashed.) After the first two messages, both the client and the server can compose the Message and compute the HashedKeyMac as the HMAC with the HashedKey. The client then sends the Key encrypted with the HashedKeyMac to the server, which decrypts the Key as $\text{KeyXorHashedKeyMac} \oplus \text{HashedKeyMac}$. In the next step, the server verifies whether $\text{hash}(\text{Key}) = \text{HashedKey}$. If this is the case, it has successfully authenticated the client. If not, the server aborts the session. At last, the server uses the Key to authenticate the same Message to the client. By also computing KeyMac, the client can verify that the last message was indeed sent by the server. Since both parties can compose the Message, the message authentication codes (MAC) can be sent without the Message. The Username is included in the Message because it wasn't authenticated in the first message. Without this, a man-in-the-middle could replace it to authenticate the user for another account where they use the same password. Let's analyze how Simplified-SCRAM is or can be made resistant to all but one of the above properties:

1. **Database compromise** thanks to salting and stretching: An attacker who compromised the server's database learns only the HashedKey but not the Key, which is required to impersonate a user. As noted above, the best way to find the preimage of the HashedKey is to guess the low-entropy Password. Due to the Salt, the Password of each user has to be attacked separately. Due to the IterationCount, the search for the Password is slowed down by several orders of magnitude.
2. **Replay attacks** thanks to the server nonce: As long as the server doesn't issue the same ServerNonce twice, earlier KeyXorHashedKeyMac values cannot be replayed because the earlier MAC doesn't match the current Message.
3. **Pooled brute-force attacks** thanks to the client nonce: Even if the ServerNonce, Salt, and IterationCount are chosen by a man-in-the-middle, KeyXorHashedKeyMac depends on the unique ClientNonce, which prevents pooled brute-force attacks.
4. **Individual brute-force attacks** thanks to a minimum iteration count: Unfortunately, the standard says only that servers should choose an IterationCount of at least 4096. It's important, however, that clients are programmed to reject an IterationCount below a certain threshold. Otherwise, a man-in-the-middle can send an IterationCount of 1, which makes it much easier to search for the Password that led to KeyXorHashedKeyMac. While this weakness can easily be addressed when writing a client, not standardizing the minimum iteration count can lead to incompatibilities between different implementations of the standard.
5. **Denial-of-service attacks** thanks to a maximum iteration count: The standard notes that a compromised server or a man-in-the-middle can perform a computational denial-of-service attack on clients by sending a big IterationCount. For this reason, clients should reject an IterationCount above a certain threshold. This threshold can be relatively high because the derived Key can be cached by the client for future authentications. This means that each client has to perform the key derivation only once. It's therefore no problem if the derivation takes several seconds.

6. **Server impersonation** thanks to mutual authentication: The KeyMac prevents an attacker from faking the authentication success. Since the KeyMac depends on the ClientNonce, the server messages cannot be replayed from an earlier session.
7. **Wrong server** thanks to binding to the domain name: We will look at the two standardized options for channel binding in the [next box](#). For now, let's imagine that a variant of SCRAM requires that the domain name of the server is appended to the Message. In other words, ChannelBinding: ServerDomain. Due to mutual authentication, a man-in-the-middle is forced to relay the communication between the client and the server. If the client connects to the wrong server, then the Message on the client is different from the Message on the actual server, which causes the authentication to fail.
8. **Compromised certification authority** thanks to binding to the server certificate: We can improve on the domain binding with ChannelBinding: hash(ServerCertificate). This prevents a man-in-the-middle from using a different certificate for the same ServerDomain, which they might obtain from a compromised certification authority.
9. **Compromised server key** thanks to binding to the session key: TLS uses a [Diffie-Hellman key exchange](#) to derive a session key, which is then used to encrypt and authenticate all messages. By choosing ChannelBinding: hash(SessionKey), we can detect a man-in-the-middle who compromised the private key of the server's certificate. Either the TLS connections from the client to the attacker and from the attacker to the server have different session keys, or the attacker can neither decrypt nor modify the communication between the client and the server. In the latter case, TLS fulfills its purpose.
10. **Comparison attacks** thanks to a user-specific salt: If I could have written the standard, the Salt would be prefixed with the user's Username in the key derivation: Key: pbkdf2(Password, Username + Salt, IterationCount). Not only does this prevent a compromised server from determining whether two accounts are protected with the same Password, it also guarantees the user that an attacker cannot run a pooled brute-force attack after compromising the database. Otherwise, a faulty server implementation, which chooses the same Salt for every user, can ruin the brute-force resistance for its users.
11. **Wrong server after database compromise** thanks to a server-specific salt: I would even go one step further and prefix the Salt also with the ServerDomain: Key: pbkdf2(Password, ServerDomain + Username + Salt, IterationCount). This prevents one service provider from impersonating the user at another service provider once the database of the latter has been compromised. Without this prefix, the former service provider can send back the Salt and the IterationCount from the compromised database and recover the Key used with the latter service provider. Since the former provider also knows the HashedKey, the mutual authentication will succeed, which makes the attack unnoticeable to the user. Another desirable benefit of this prefix is that it forces different servers to use separate authentication databases. For example, an attacker who compromised the outgoing mail server would no longer be able to retrieve the user's mail at the incoming mail server. These precautions only make sense, though, if the Password is never shared with the server, not even when setting the password. This means that the client has to choose the Salt and the IterationCount and then generate the string Salt + IterationCount + HashedKey so that the user can paste it into the account configuration interface. For this to work, setting and replacing the password would have to be standardized as well.
12. **User impersonation after server compromise**: SCRAM is not resistant to a server compromise. If an attacker manages to control the server (or alternatively to compromise the database and to intercept the communication channel), they learn the Key, which is all that is needed to impersonate the user. In order to prevent this, we need [public-key cryptography](#).

The remaining boxes in this subsection just add more context. The conclusion of this little detour is the same as the conclusion of the whole article: We could do so much better if we only wanted to (and were better informed). The standards exist, we just need to deploy them...

▼ TLS channel bindings (SCRAM-PLUS)

Channel binding is discussed in [section 6 of RFC 5802](#). If a server supports channel binding, it advertises the authentication mechanism as SCRAM-<hash-function>-PLUS. An example is SCRAM-SHA-256-PLUS as specified in [RFC 7677](#). Since mutual authentication is established on the application layer by SCRAM, the security layer has to provide only message confidentiality and message authentication but not [party authentication](#) when channel binding is used. As a consequence, SCRAM-PLUS can be used without a [public-key infrastructure](#), which means that servers can use [self-signed certificates](#). Binding the application layer to the security layer doesn't change the security layer. A TLS implementation needs to be changed only if it doesn't allow the application layer to access the necessary values.

[RFC 5929](#) defines three different channel bindings for TLS, where only two of them are relevant for us:

- [tls-server-end-point](#) uses the hash of the server's certificate: hash(ServerCertificate). The advantage of this binding is that it can easily be used with a [reverse proxy](#). Its disadvantage is that it doesn't protect against compromised server keys.
- [tls-unique](#) uses the first TLS Finished message of the latest [TLS handshake](#). Since the Finished message contains a hash over all previous handshake messages, it uniquely identifies a particular TLS connection. For full TLS handshakes, the first Finished message is sent by the client. For abbreviated TLS handshakes, the first Finished message is sent by the server. Depending on which type of handshake has been performed and which of the two endpoints you implement, you have to call either [getFinished\(\)](#) or [getPeerFinished\(\)](#) to access the [right message](#) for channel binding. In theory, [tls-unique](#) is the preferred option for channel binding because it also prevents attacks with [compromised server keys](#). In practice, however, [tls-unique](#) requires proxy servers to forward the first Finished message to the application server so that it can compose the [SCRAM Message](#) correctly, which makes this option more difficult to deploy.

▼ Authentication on the Web

Given the many desirable properties of SCRAM, you might wonder whether we can also use this mechanism when logging in to websites. The short answer is yes: Apart from channel binding, you can implement SCRAM with JavaScript in the browser. The longer answer is no: Since users cannot trust the code that is loaded by a website, nothing is gained by implementing SCRAM for logging in to your website. The most desirable property for authentication mechanisms on the Web is to prevent phishing, where a victim is tricked to connect to a wrong server. Since in this case the code is loaded directly from the attacker, it can send your password directly to the attacker. The only way to make password-based authentication on the Web secure is to move the functionality from a webpage to the browser and to expose it through an API. This could be achieved with a SCRAM-SHA-256-PLUS extension to HTTP authentication, where the browser takes care of the authentication messages and the server sets the session cookie on success. This is not likely to happen anytime soon, though. The trend goes rather towards replacing or supplementing password-based authentication with public-key cryptography, for example with the Web Authentication (WebAuthn) standard. None of this is a problem for mail clients, though, since their code isn't loaded from untrusted sources.

▼ Password-authenticated key exchange (PAKE)

The goal of key exchange protocols is to establish a shared secret between two parties, which can then be used to encrypt and authenticate all messages between them. In order to ensure that the secret is shared between the intended parties, they need to authenticate themselves initially. Otherwise, a man-in-the-middle can establish one secret with the first party and another secret with the second party, allowing them to intercept all messages between the two parties. One way of achieving this is by relying on third parties, so-called certification authorities, to confirm the identity and the public key of a party. Another way of achieving this is by relying on a secret that they already share. Password-based authentication mechanisms such as SCRAM accomplish this by binding to the secure channel after it has already been established. Password-authenticated key exchange (PAKE) protocols, on the other hand, accomplish this by using the password during the key exchange for mutual authentication. You don't want to use a key derived from the password as the shared secret because once the password is compromised, all earlier sessions are compromised as well. Password-authenticated key exchange protocols avoid this by using public-key cryptography to establish a secret which is unique to each session and cannot be derived from the password. One example is the Secure Remote Password (SRP) protocol. Not only does it achieve the just-mentioned property, which is called forward secrecy, it's also resistant to user impersonation after server compromise: The server never learns the necessary information to impersonate its users. TLS supports SRP as a key exchange algorithm under the label TLS-SRP but just like SCRAM it seems to be rarely used. One downside of SRP is that it leaks the username to any eavesdropper during its TLS handshake.

Access protocols

Besides proprietary protocols, most incoming mail servers allow mail clients to access the user's mailbox with POP3 or IMAP. If your mail client and your mail server support both protocols, you should choose the latter as it's much more powerful. The main reason for including POP3 in this article is that it's much easier to use from the command-line interface.

▼ Communication logging in Apple Mail

Apple Mail allows you to inspect its communication with your mail servers by clicking on "Connection Doctor" in the "Window" menu and then on "Show Detail". You can also enable "Log Connection Activity" there to persist the log of its communication in the folder `~/Library/Containers/com.apple.mail/Data/Library/Logs/Mail/`. Since the log files include the content of all your messages, including deleted ones and those of removed accounts, you should enable this option only if you really need it.

▼ Communication logging in Thunderbird

You can inspect how Thunderbird interacts with your mail servers by logging its communication with the following commands:

Operating system: Logging level:

```
$ export MOZ_LOG=timestamp,append,SMTP:3,POP3:3,IMAP:3
$ export MOZ_LOG_FILE=~/.Downloads/thunderbird.moz_log
$ /Applications/Thunderbird.app/Contents/MacOS/thunderbird-bin
```

Enter the above commands in your command-line interface, then open the log file in a text editor, such as Visual Studio Code.

Post Office Protocol Version 3 (POP3)

The Post Office Protocol Version 3 (POP3) is specified in RFC 1939. Similar to ESMTP, POP3 is a text-based application-layer protocol, which can be used with Implicit TLS or with Explicit TLS. POP3 with Implicit TLS is also known as POP3S. Just like SMTP, POP3 commands consist of four letters and an extension mechanism was introduced after the initial release of the standard. After authenticating the user, POP3 allows the client to list, retrieve, and delete messages. POP3 is designed to move messages from a remote queue into a local queue. It doesn't support read statuses, mailbox folders, message uploads, or partial fetches.

The following POP3 tool works in the same way as the ESMTP tool above. Most of the remarks I made earlier therefore still apply. In particular, I advise you to use it only with accounts created for this purpose. The tool uses Thunderbird's configuration database and Google's DNS API to resolve the server you want to connect to. Copy the commands in bold to your command-line interface by clicking on them. The text in gray mimics what the responses from the server look like. The actual responses will be different. Each response starts with either +OK or -ERR. The former indicates that your command was successful, the latter indicates that an error occurred. If necessary, you can always kill the current process and thereby the connection by pressing ^C (control + c). If you use Gmail, you have to enable POP3 access in your account settings and allow access from insecure apps.

Address:	<input type="text" value="alice@example.org"/>	Username:	<input type="text" value="Full address"/>
Password:	<input type="password" value="Your password"/>	Credential:	<input type="text" value="Plain password (PLAIN)"/>
Security:	<input type="text" value="Implicit TLS"/>	Challenge:	<input type="text" value="<unique@example.org>"/>
Server:	<input type="text" value="pop3.example.org"/>	Delete:	<input type="checkbox"/>
Port:	<input type="text" value="995"/>	<input type="button" value="↶"/> <input type="button" value="🗑"/> <input type="button" value="↷"/>	

```
$ openssl s_client -quiet -crlf -connect pop3.example.org:995
+OK Implementation ready.
USER alice@example.org
+OK
PASS
+OK Logged in.
LIST
+OK 1 messages:
1 623
.
RETR 1
+OK Message follows:
{HeaderAndBody}
.
QUIT
+OK Logging out.
```

▼ POP3 commands

All commands are case-insensitive and must be terminated with CR+LF. Responses spanning several lines are terminated by a period on a line of its own. If a line starts with a period, an additional period is prepended to the line. After user authentication, the server enumerates all messages in the inbox sorted by their date, where 1 is assigned to the newest message. All message numbers are expressed in the decimal system. The mapping between numbers and messages is valid only for the duration of the session. To ensure that the numbers remain valid and that the messages remain available for the duration of the session, the server locks the mailbox. If the server fails to acquire the lock because the same mailbox is being accessed simultaneously, it responds with -ERR to the last authentication command. As long as the server can guarantee consistency for each client, it can allow simultaneous access. All sizes are specified in bytes. POP3 servers must support the following commands:

Command	Argument	Response	Description
<u>USER</u>	Username	-	Indicate the user whose messages shall be retrieved.
<u>PASS</u>	Password	-	Transmit the password to authenticate the user.
<u>STAT</u>	-	Count Size	Return the count and size of all messages.
<u>LIST</u>	[Number]	Number Size	List the size of all messages [or of the specified one].
<u>RETR</u>	Number	Message	Retrieve the message with the given number.
<u>DELE</u>	Number	-	Mark the message with the given number as deleted.
<u>RSET</u>	-	-	Unmark all messages that were marked as deleted.
<u>NOOP</u>	-	-	<u>Do nothing</u> besides <u>keeping the connection alive</u> .

Command	Argument	Response	Description
<u>QUIT</u>	-	-	Delete the marked messages and close the connection.

The mandatory commands of POP3. (The USER and PASS commands are strictly speaking optional.)

▼ POP3 extensions

[RFC 2449](#) defines an extension mechanism for POP3. It introduces the CAPA command, to which the server responds with the supported capabilities. If a server doesn't recognize an optional command, such as CAPA, it responds with -ERR. Each line in the response to the CAPA command indicates a command that the client can use or a behavior which the client should know about:

Command	Argument	Response	Description
<u>CAPA</u>	-	Capabilities	List the supported capabilities.
<u>STLS</u>	-	-	Upgrade the connection from TCP to TLS just like <u>STARTTLS</u> .
<u>TOP</u>	Number X	Message	Return the header and the top X body lines of the specified message.
<u>UIDL</u>	[Number]	Number ID	List the permanent ID of all messages [or just the specified one].

Some optional commands of POP3. These commands extend the basic functionality of POP3.
Unlike the message numbering, the IDs are guaranteed to stay the same across sessions.

Behavior	Argument	Description
<u>PIPELINING</u>	-	Indicates that the server can handle <u>multiple commands</u> at a time.
<u>RESP-CODES</u>	-	Indicates that the server supports <u>extended response codes</u> in square brackets.
<u>AUTH-RESP-CODE</u>	-	Indicates that the server tells the client why an authentication attempt failed.
<u>IMPLEMENTATION</u>	Name	Indicates the name of the server's POP3 implementation for troubleshooting.
<u>SASL</u>	Mechanisms	Indicates the <u>SASL mechanisms</u> which can be used with the <u>AUTH command</u> .
<u>LOGIN-DELAY</u>	Seconds	Indicates how many seconds the client has to wait before connecting again.
<u>EXPIRE</u>	Days	Indicates after how many days the server deletes (retrieved) messages.

The server can indicate additional behavior in its response to the CAPA command.
LOGIN-DELAY and EXPIRE allow the server to conserve its resources.

```
$ telnet mail.gandi.net 110
Trying 217.70.178.9...
Connected to mail.gandi.net.
Escape character is '^]'.
+OK Dovecot ready.
CAPA
+OK
CAPA
TOP
UIDL
RESP-CODES
PIPELINING
AUTH-RESP-CODE
STLS
USER
SASL PLAIN
.
QUIT
+OK Logging out
Connection closed by foreign host.
```

Just like STARTTLS, STLS is advertised only in TCP connections. Gmail doesn't support POP3 with Explicit TLS but Gandi does.

▼ APOP authentication

To the best of my knowledge, APOP stands for Authenticated Post Office Protocol. It's a challenge-response authentication mechanism similar to CRAM-MD5 with the same properties. Even though APOP is an optional command, it's not advertised in the response to the CAPA command because a POP3 server already indicates support for the APOP command by including the challenge in its initial greeting. The Challenge is of the form <Nonce@Host>. The Response is the hexadecimal encoding of md5(Challenge + Password), where MD5 is a cryptographic hash function. You find an example session in RFC 1939.

Internet Message Access Protocol (IMAP)

The Internet Message Access Protocol (IMAP) is specified in RFC 3501. IMAP works similar to ESMTP and POP3, it just has many more commands and options. An IMAP mailbox acts as a remote drive for messages instead of files, where the drive is being shared among several clients. IMAP allows users to create, delete, and rename folders, to upload and move messages between them, to mark messages as read or as flagged, to search the mailbox remotely, and to download messages without their attachments.

The following IMAP tool works just like the ESMTP and POP3 tools above. As you might mess up your mailbox or delete messages you still wanted by accident, you should run the following commands on test accounts only. If you want to use your real account, you do so at your own risk. Certain commands have side effects, such as marking messages as read. Make sure you fully understand a command before using it. This tool also uses Thunderbird's configuration database and Google's DNS API to resolve the server you want to connect to. Neither IMAP nor the tool are self-explanatory. You find more information in the tooltips and the boxes below.

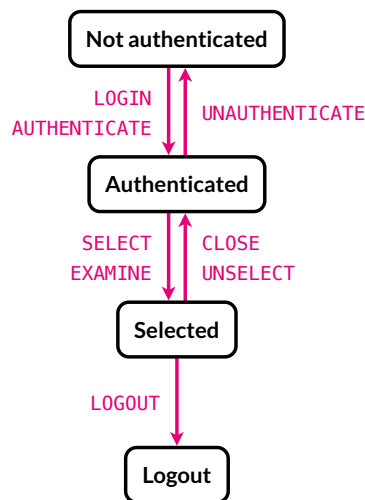
Address:	<input type="text" value="alice@example.org"/>	Search:	<input type="checkbox"/>
Password:	<input type="text" value="Your password"/>	Messages:	<input type="text" value="4"/>
Security:	<input type="text" value="Implicit TLS"/>	Criterion:	<input type="text" value="Subject"/>
Server:	<input type="text" value="imap.example.org"/>	Value:	<input type="text"/>
Port:	<input type="text" value="993"/>	Date:	<input type="text" value="13.05.2021"/>
Username:	<input type="text" value="Full address"/>	Delete:	<input type="checkbox"/>
List:	<input type="checkbox"/>	Append:	<input type="checkbox"/>
Write:	<input type="checkbox"/>	Idle:	<input type="checkbox"/>
Fetch:	<input type="text" value="Date"/>		<input type="button" value="↶"/> <input type="button" value="🗑"/> <input type="button" value="↷"/>

```
$ openssl s_client -quiet -crlf -connect imap.example.org:993
* OK Implementation ready
I LOGIN alice@example.org
I OK LOGIN completed
E EXAMINE "INBOX"
* 4 EXISTS
* 2 RECENT
* OK [UNSEEN 3]
* OK [UIDNEXT 5]
* OK [UIDVALIDITY 1]
* OK [PERMANENTFLAGS ()]
* FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
E OK [READ-ONLY] EXAMINE completed
F FETCH 4 ((BODY[]))
* 4 FETCH ((BODY[])) {123}
{Data}
)
F OK FETCH completed
O LOGOUT
* BYE Logging out
O OK LOGOUT completed
```

After the initial greeting by the server, the client sends commands, to which the server responds. Since multiple commands can be in progress at the same time, the client tags each command with a unique identifier, such as A, B, C, or a dot .. The server prefixes each line of its response with * and completes its response with a line which starts with the tag chosen by the client. The tag is followed by a status response: OK for success, NO for failure, or BAD for protocol errors. Don't worry about reusing tags in a single session, you can run a command repeatedly with the same tag. If you want to fetch another message, for example, just enter another message number and copy the generated command again. If you use Gmail, you have to enable IMAP access in your account settings and allow access from insecure apps.

▼ Protocol states

Most IMAP commands can be called only in certain states. (The same is true for POP3 but I didn't deem it worth mentioning.) Unless the connection has been pre-authenticated, the IMAP protocol starts in the not-authenticated state. Before the client can do anything else, it has to issue a LOGIN or AUTHENTICATE command. (When using Explicit TLS, the client can also send the STARTTLS request.) While the LOGIN command is followed by the username and the password, AUTHENTICATE can be used with any SASL mechanism which is supported by the server. If the user has been authenticated successfully, the protocol enters the authenticated state. The client has to SELECT or EXAMINE a folder before it can issue commands that affect existing messages. Once in the selected state, the client can SEARCH and FETCH messages (among other things). The client can issue the LOGOUT command in any state, which takes the protocol to the logout state, in which the server closes the connection. If you want to inspect, modify, or delete messages in a different folder, you can CLOSE or UNSELECT the current folder and open another one. The difference between these two commands is that the former removes messages marked for deletion permanently while the latter does not. (UNSELECT is an extension, which can be used only if the server supports it.) By using the new UNAUTHENTICATE command, which not many servers support yet, the client can authenticate as a different user without having to re-establish the TCP and TLS connection. Here is a simplified version of the official state diagram:



The protocol states and how to transition between them.
 LOGOUT can be called in any state except the logout state.
 UNAUTHENTICATE can also be called in the selected state.

A word on terminology: The standard and some mail clients, such as Apple Mail, speak of mailboxes rather than folders. When I speak of mailboxes, I usually refer to the mail account as a whole. Thunderbird, on the other hand, avoids the term completely. I mostly ignore IMAP folders and how to CREATE, DELETE, and RENAME them. The only important aspect for us is that INBOX is a special name and always refers to the primary folder of the user.

▼ Data formats

While IMAP is also mostly a text-based protocol, it's more difficult to read and to write for humans than SMTP and POP3. This is due to the various data formats it uses, which are defined in section 4 of RFC 3501. We're interested in just three of them:

- **String:** Strings are either unquoted, quoted, or prefixed with their length. Prefixing the length has the advantage that the string doesn't have to be escaped. In particular, no periods have to be added to transmit a message. This technique turns IMAP into a binary protocol temporarily, making it difficult for humans. Let's look at an example for each string variant.

```
E EXAMINE INBOX
```

Unquoted string: Since INBOX contains no spaces, it doesn't have to be quoted (but it can be).

```
E EXAMINE "Sent Mail"
```

Quoted string: Since Sent Mail contains a space, it has to be quoted. Otherwise, Sent Mail constitutes two arguments instead of one.

```
F FETCH 1 (BODY.PEEK[HEADER.FIELDS (SUBJECT)])
* 1 FETCH (BODY[HEADER.FIELDS (SUBJECT)] {20}
Subject: Example

)
F OK FETCH completed
```

Length-prefixed string: If a string contains newline characters, its length in bytes has to be prefixed in curly brackets. Since each ASCII character is encoded in a single byte and newlines consist of two characters, namely carriage return (\r) and line feed (\n), Subject: Example\r\n\r\n consists of 20 bytes. (IMAP includes an empty line after the header and the length-prefixed string is part of a list.) The standard calls this the literal form. When a literal string is transmitted from the client to the server, the client has to wait for a continuation response from the server after sending the length. Activate Write and Append in the tool above to see an example.

- **Lists:** Lists are used when a variable number of items are to be transmitted. A single space is used to separate adjacent items and the list is enclosed by parentheses. Lists can be nested in other lists and lists can be empty. Let's look at two examples:

```
F FETCH 1 (FLAGS)
* 1 FETCH (FLAGS (\Seen))
F OK FETCH completed
```

Nested list: The response contains a nested list of flags.

```
F FETCH 1 (FLAGS)
* 1 FETCH (FLAGS ())
F OK FETCH completed
```

Empty list: When the message hasn't been seen yet, the nested list is empty.

- **Nil:** NIL indicates that an item doesn't exist. You have to consult the formal syntax to see where NIL is allowed.

▼ Message numbers

Similar to POP3, messages in IMAP can be referenced either by their position in a folder or by their unique identifier (UID):

- **Position:** If the response to the SELECT or EXAMINE command says with 8 EXISTS that 8 messages exist, then 1 refers to the oldest message and 8 to the newest message. All numbers in between are guaranteed to refer to messages as well. When a message is removed from the folder, the position of all subsequent messages is decremented by one. Messages are always added at the end of the list: When a new message is added to the 8 existing ones, it can be referenced by the number 9.
- **UID:** UIDs are numbers which are assigned in ascending order to messages. Unlike the position of a message, which can change within and across sessions, its UID is meant to stay the same. When a message is deleted, the UIDs of subsequent messages don't change. As a consequence, UIDs are not necessarily contiguous. Mail clients use UIDs to synchronize flags and deletions of the messages they've already retrieved from the server. IMAP has a special UID command, which allows the client to use SEARCH, FETCH, STORE, and COPY with UIDs instead of positions. For example, clients issue the command TAG UID FETCH 1:{LastSeenUIDNEXT-1} FLAGS to discover changes to old messages according to the informational RFC 4549. In other words, clients find out which messages have been deleted while they were offline by fetching the flags for all locally stored messages from the server every time they reconnect. All messages whose UID is no longer in the response are then removed. If the UIDNEXT value in the response to EXAMINE or SELECT is bigger than the last time the client connected, the client knows that new messages arrived in the meantime. If the UIDVALIDITY value in the same response is bigger than the last time it connected, the client has to invalidate its UIDs and rebuild its database. Due to the overhead this causes, servers should avoid invalidating UIDs. However, since folders can be renamed and clients reference them by name, the content of a folder can change completely. By using the current timestamp as the UIDVALIDITY value whenever a folder is created or renamed, servers can force clients to refetch all messages in such a folder.

▼ Message sets

FETCH, STORE, and COPY operate on a set of messages. You can specify a single number, such as 4, a range of numbers, such as 6:8, or a combination thereof, such as 4,6:8. When referencing messages by their position, 6:8 is guaranteed to select three messages as long as there are at least eight messages in the folder. When using the UID command, 6:8 selects between zero and three messages, depending on whether messages with UIDs in this range have been deleted. * represents the largest number in use. When referencing messages by their position, * corresponds to the number of messages in the folder. If the folder is empty, you get an error when using *. If you want to fetch the flags of all messages, you can use F UID FETCH 1:* (FLAGS). If you want to fetch all new messages, you can use F UID FETCH {LastSeenUIDNEXT}:* (FLAGS BODY.PEEK []). {LastSeenUIDNEXT} needs to be replaced with an actual number, of course. (You have to replace the curly brackets with an actual value in all my examples except when the curly brackets are used as the length prefix of a literal string.)

▼ Message flags

IMAP messages can be tagged with labels, which are called flags. Most flags are persisted across sessions but some flags are applied only within a session. Flags defined by IETF standards start with a backslash. RFC 3501 defines the following flags:

- \Seen: The message has been seen (i.e. read).
- \Answered: The message has been answered.
- \Flagged: The message is flagged for special attention.
- \Deleted: The message is marked for deletion by CLOSE or EXPUNGE.
- \Draft: The message is marked as a draft, i.e. it hasn't been sent yet.
- \Recent: The message has arrived in the folder recently. This flag cannot be set or removed by the client. If the client uses SELECT instead of EXAMINE, this flag is no longer set in later sessions.

In the response to the EXAMINE or SELECT command, the server includes the FLAGS which are defined in the folder. As part of the PERMANENTFLAGS response, the server indicates which of the flags the client can set and remove. If the list includes *, the client can create custom tags, which may not start with a backslash. The formal syntax specifies the permissible characters.

S STORE 1 +FLAGS (custom-flag)

```
* FLAGS (\Answered \Flagged \Deleted \Seen \Draft custom-flag)
* OK [PERMANENTFLAGS (\Answered \Flagged \Deleted \Seen \Draft custom-flag \*)]
* 1 FETCH (FLAGS (\Seen custom-flag))
S OK STORE completed
```

How a custom flag can be created and set if the IMAP server supports it.

▼ Internal date

Besides flags, messages have other attributes as well. One of them is the internal date, which records when the message was received. Mail clients can display messages with this date instead of the sender-chosen origination date. Since Apple Mail also displays the received date instead of the sent date when fetching messages via POP3, it seems to rely on the Received header field indeed. Other attributes which can be fetched are the message size and the body structure of multipart messages.

F FETCH 1 (INTERNALDATE)

```
* 1 FETCH (INTERNALDATE "24-Nov-2020 15:43:32 +0000")
F OK FETCH completed
```

How to fetch when a message was received.

▼ IMAP commands

Some of the commands used in the above tool benefit from additional information. This is what you should know about them:

- EXAMINE vs. SELECT: Both commands open a folder in order to search and fetch the messages in it. The difference is that EXAMINE opens the folder in read-only mode, while SELECT also allows the client to change and delete messages. This is made visible in the response line which starts with the tag: It contains either [READ-ONLY] or [READ-WRITE].
- SEARCH: Saving a search result for later operations requires the SEARCHRES extension. If your IMAP server doesn't support it, you have to search without the RETURN (SAVE) part: S SEARCH {Criterion}. The server then returns the positions of all the messages that match the criterion: * SEARCH 2 5 8. Search criteria can also be combined: S SEARCH {Criterion1} {Criterion2} {etc.}. IMAP also supports the logical operators NOT and OR besides the implicit "and": NOT {Criterion} and OR {Criterion1} {Criterion2}. As you can see, the query language of IMAP is quite powerful.
- FETCH: The first argument to the FETCH command is a set of messages. If the server supports saving the search result with RETURN (SAVE), you can alternatively reference the search result with the dollar sign. The second argument is a list of the data attributes you want to fetch. The difference between BODY [{Section}] and BODY. PEEK [{Section}] is that the former sets the \Seen flag while the latter does not. You can use either one to fetch the desired section of the specified messages.
- STORE: The STORE command allows the client to alter the flags of a message. Similar to FETCH, the first argument is either a message set or \$ for a search result. After that, you can replace the flags of the messages with FLAGS ({NewFlags}), add additional flags to the existing flags with +FLAGS ({FlagsToAdd}), or remove some flags from the existing flags with -FLAGS ({FlagsToRemove}). Messages are deleted by flagging them as \Deleted and then using the CLOSE or EXPUNGE command. The former also closes the folder and takes you back to the authenticated state, whereas the latter doesn't do that.
- APPEND: Mail clients use this command to store sent messages in the user's mailbox. Since the target folder is specified in the first argument, this command can be used from the authenticated state. Besides the flags you want to set on the appended message, you can also specify the internal date in an optional third argument. The fourth argument is the message that you want to append, which has to be

transmitted as a length-prefixed string. Since counting bytes manually is a hassle, the tool does the counting for you when you enable Write and Append. You can edit the used message in the ESMTP tool above.

▼ IMAP extensions

Given the importance of IMAP in the email ecosystem, there are numerous extensions for it. You can query which extensions a server supports with the CAPABILITY command. You see an example when you enable the Search or the Idle option in the tool above. Before using the enabled commands, make sure that your server supports the listed extensions. Issuing C CAPABILITY is often not necessary since many IMAP servers list their capabilities automatically in their response to the LOGIN command.

The most important extensions to IMAP are (ignoring the ones for internationalization, such as support for UTF-8):

- IMAP4REV1 (RFC 3501): By listing this among its capabilities, a server indicates that it supports IMAP version 4 revision 1 as published in 2003. IMAP4rev1 is the protocol we've been discussing in this article. A second revision, which adds most of the extensions mentioned here to the core protocol, is in the making. The changes to the first revision are listed in its appendix.
- STARTTLS (RFC 2595): This extension allows the client to upgrade the connection from TCP to TLS with . STARTTLS. You have to use telnet {ServerDomain} 143 to see this capability listed by the server. (143 is IMAP's port for Explicit TLS.)
- SASL-IR (RFC 4959): If the server has this capability, the client can append its initial SASL response to the AUTHENTICATE command, which saves one round trip. Example: . AUTHENTICATE PLAIN {Base64EncodingOfUsernameAndPassword}.
- ENABLE (RFC 5161): While CAPABILITY allows the server to list the extensions it supports, the ENABLE command allows the client to list the extensions it supports. This allows the server to send unsolicited responses defined by these extensions.
- ID (RFC 2971): For improving bug reports and assembling usage statistics, it's useful to know which implementation of the protocol the other party uses. The ID command allows the client to send a list of key-value pairs to the server and receive a list of key-value pairs in return. Some keys are specified in the RFC but any string of at most 30 bytes can be used as a key. For example, a client can send TAG ID ("name" "ef1p") to the server and receive * ID ("name" "Dovecot") in return.
- IDLE (RFC 2177): Instead of regularly polling the server for changes, a client can instruct the server with the IDLE command to transmit changes to the current folder in real time. You can enable Idle in the tool above to see an example. As long as the TCP connection between the client and the server remains open, the client is notified about new messages immediately. In order to avoid timeouts due to inactivity, the client can send the NOOP command, which does nothing, from time to time.
- ESEARCH (RFC 4731): ESEARCH is an extension to the SEARCH and UID SEARCH commands, which allows the client to choose between several result options by issuing . SEARCH RETURN ({Options}) {Criteria}. The options are MIN to return the position or UID of the first message in the folder which satisfies the criteria, MAX to return the position or UID of the last message in the folder which satisfies the criteria, COUNT to return the number of messages in the folder which satisfy the criteria, and ALL to return the numbers of all messages which satisfy the criteria. When using the ALL option, the messages are returned in the set syntax instead of the space-separated enumeration of all messages. For example, a client can query how many messages are flagged with TAG SEARCH RETURN (COUNT) FLAGGED.
- SEARCHRES (RFC 5182): SEARCHRES is an extension to the ESEARCH extension. Any server which supports SEARCHRES also has to support ESEARCH. SEARCHRES adds the result option SAVE, which tells the server to save the search result for later use instead of returning it. The client can reference the search result with \$ in the FETCH, STORE, and some other commands. One advantage of this is that the client doesn't have to wait for the search result before it can submit a subsequent command.
- UIDPLUS (RFC 4315): UIDPLUS adds the command UID EXPUNGE and additional response codes, which inform the client about the UID of an appended or copied message. This is useful for clients to synchronize with servers more efficiently.
- CONDSTORE (RFC 4551): CONDSTORE is by far the biggest extension in this list. It introduces MODSEQ as an additional message attribute and HIGHESTMODSEQ as an additional response to the EXAMINE and SELECT commands. MODSEQ works like UID but instead of assigning a permanent, strictly increasing number to each message, it assigns a permanent, strictly increasing number to each message modification. By remembering the HIGHESTMODSEQ value to which they synchronized, clients can use the extended STORE or UID STORE commands to modify messages on the server only if no other client modified them in the meantime. (CONDSTORE stands for conditional STORE.) CONDSTORE also extends other commands. For example, clients can use the CHANGEDSINCE modifier to fetch changes to messages more efficiently. Instead of fetching the flags of all messages every time they connect, clients can fetch the flags of just the messages which changed since the last time: TAG UID FETCH 1:* (FLAGS) (CHANGEDSINCE {LastSeenHIGHESTMODSEQ}). Unfortunately, clients can't detect message deletions like this.
- QRESYNC (RFC 5162): QRESYNC extends CONDSTORE to allow for quick mailbox resynchronization but it's rarely supported. By remembering the UIDs of expunged messages with the corresponding MODSEQ value, servers can inform clients efficiently about deleted messages. Both CONDSTORE and QRESYNC were updated in RFC 7162. In the absence of QRESYNC, clients can perform a "binary" search to find the first message whose position changed. Clients need to do this only when the EXISTS count from the server is different from the local count after adding all the newly arrived messages. Clients can retrieve the UIDs of several messages at once by issuing TAG UID SEARCH {Position1}, {Position2}, {etc.}.

- **CHILDREN (RFC 3348)**: This extension allows the server to indicate in its response to the LIST command whether a folder \HasChildren or \HasNoChildren. This allows the client to display a folder as expandable without having to query for potential children with additional requests.
- **SPECIAL-USE (RFC 6154)**: Folders often have a specific purpose such as storing sent or deleted messages. This extension allows clients to inform each other about the special use of a folder without having to rely on specific names for the folders and without having to ask the user where to store specific messages. The defined purposes are \All, \Archive, \Drafts, \Flagged, \Junk, \Sent, and \Trash. The purpose can be set when creating a new folder and is returned in the response to the LIST command. Gmail supports this extension and its response is roughly what you see in the tool above.
- **NAMESPACE (RFC 2342)**: This extension introduces a NAMESPACE command, which allows clients to discover the namespaces of personal folders and of shared folders. My understanding is that this is mostly used in corporate settings.
- **MOVE (RFC 6851)**: This extension defines the commands MOVE and UID MOVE to move messages from one folder to another. When MOVE is not supported, clients have to COPY messages to another folder and then delete the copied messages in the old folder with STORE and EXPUNGE. This is inefficient for both the client and the server and can lead to undesirable side effects.
- **QUOTA (RFC 2087)**: This extension allows clients to get and set the storage quota of their mailbox. For example, when using Q GETQUOTA "" in my Gmail test account, I get * QUOTA "" (STORAGE 145 15728640). The first number is the current usage in kibibytes, the second number the resource limit, which matches the 15 GB of free storage as advertised by Google. When using Q SETQUOTA "" (STORAGE 1000), I get Q NO [CANNOT] Permission denied. (Failure). "" denotes the so-called quota root, which allows different folders to share the same resource limit.

In addition to the extensions which are standardized by IETF, email service providers are free to define their own extensions. According to the IMAP standard, the name of an experimental or independent extension has to start with an X. For example, Gmail's custom extension is advertised as X-GM-EXT-1 in the response to the CAPABILITY command. Among other things, it allows clients to use Gmail's search syntax and Gmail's message ID.

JSON Meta Application Protocol (JMAP)

Over the last forty years, email in general and IMAP in particular became a patchwork of extensions. Given the complexity and the varying support of these extensions, writing a mail client is much more difficult than it should be. While there are efforts to unify the patchwork somewhat, there has also been a fresh start over the last couple of years. An IETF working group designed a modern protocol for client to server interaction: The JSON Meta Application Protocol (JMAP). JSON itself stands for JavaScript Object Notation, which is a popular format for storing and exchanging human-readable data. JMAP is specified in RFC 8620 and it can be used for more than just email. The data model for synchronizing email is specified in RFC 8621. If you don't like the RFC formatting, you can also read the two standards here and here.

JMAP is designed to be interoperable with IMAP mailboxes and thus shares the concepts of folders and flags with IMAP. The protocol itself, however, is completely new and addresses the following shortcomings of IMAP (and message submission):

- **Permanent identifiers**: JMAP servers assign permanent identifiers to all objects. In the case of messages, these identifiers can no longer be invalidated and they no longer change when a message is moved from one folder to another. In the case of folders, JMAP clients can detect when a folder has been renamed and no longer need to fetch all the messages in it again.
- **Efficient synchronization**: JMAP provides a simple method for getting the identifiers of created, updated, and destroyed messages and folders. As we have seen above, synchronizing a mailbox with IMAP is easy only if you stay connected to the server, which isn't an option for mobile clients.
- **Push mechanism**: In order to be informed immediately about changes to a folder, such as newly arrived messages, IMAP clients use the IDLE command. If they want to be informed about changes to several folders, they have to open a separate connection for each folder. JMAP, on the other hand, allows clients to subscribe to all changes on the server at once. Clients which can keep a connection to the server open can subscribe via the EventSource interface. Other clients, such as those on mobile phones, can register a callback URL, which allows them to use their platform-specific push technology.
- **Batching of chained commands**: When the IMAP server doesn't support certain extensions such as SEARCHRES, IMAP clients often need to wait for the response to one command before they can construct the followup command. JMAP allows clients to batch several commands and to reference the results from earlier commands in the same request. Doing so avoids round trips and makes updates more atomic (i.e. it becomes less likely that only some of the issued commands are being executed).
- **Widespread data format**: JMAP data doesn't have to be encoded as JSON and future standards can specify other data formats. The same is true for the transport protocol: While JMAP currently uses HTTPS as its transport protocol, other protocols can be added in the future. The choice of JSON and HTTPS is mostly due to their widespread adoption: There are suitable libraries for all relevant programming languages and software engineers know how to use those. It's worth mentioning that JMAP doesn't wrap binary data in JSON. Binary data is exchanged in separate connections.
- **Complexity on server**: JMAP moves the complexity of handling email's message format from the client to the server. While clients can still fetch the raw message if needed, for example when implementing end-to-end security, the server has to deal with multipart messages, content encodings, line-length limits, etc. Clients can download and upload messages as a simple JSON object. Please note that this affects neither how

messages are stored on servers nor how they are relayed to others. It just relieves programmers who want to integrate email from having to take care of encoding and decoding messages correctly.

- **Message submission:** The previous point only makes sense if clients can also submit messages for delivery in the same format. If the JMAP server supports submission, a client can instruct it to send a stored message to its recipients. The client can generate the envelope itself or let the server do it. By first storing the message as a draft and then moving it to the sent folder after sending it (see this example), JMAP also solves the double-submission problem.
- **Flood control:** Since it's not always possible to anticipate how much data the server will send back, JMAP lets clients restrict the size of responses. This feature is especially valuable on devices with limited bandwidth or expensive roaming.

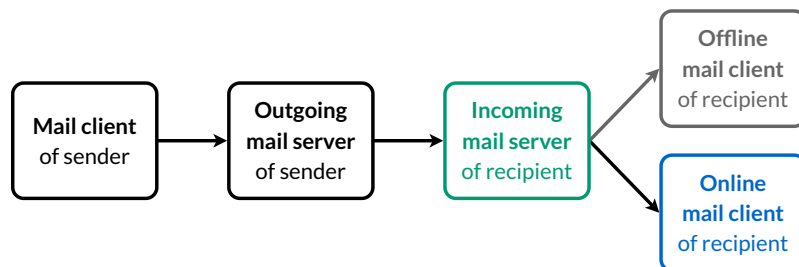
Support for JMAP is still quite rare, which is not surprising given that the standard was published only in 2019. We yet have to see whether it will become a relevant protocol for accessing one's mailbox. I certainly hope so but email is really resistant to innovation.

Email filtering

It can be useful to filter incoming messages according to custom rules. For example, you may want to move certain messages to a certain folder, mark certain messages as read, or delete certain messages automatically. Most mail clients allow their users to configure such rules, which are executed when the mail client receives a new message. There are several advantages of filtering incoming mail on the server rather than on the client, though:

- **Synchronization:** If the filtering rules are stored on the incoming mail server, they can be inspected and edited through any of the user's mail clients. Otherwise, users have to remember on which client they've created the rule that they want to modify now.
- **No race conditions:** If the filtering rules are stored on a mail client, then the rules are not applied when this mail client is offline. In this situation, other mail clients see unfiltered messages. If these mail clients apply rules of their own, you might run into race conditions, where the order in which clients see incoming messages determines the outcome of the filtering.
- **Rules for absence:** Some rules, such as sending out-of-office replies, shall run precisely when all mail clients are offline. This is not possible when the rules must be executed by mail clients.
- **Rejection during delivery:** Unlike clients, incoming mail servers can reject a message during its delivery. By sending the 550 response code during the SMTP session, the incoming mail server can inform the sender about the rejection without causing backscatter with bounce messages.

To achieve server-side filtering, we need a standardized mail filtering language and a standardized filter management protocol.



How a message is delivered from the mail client of the sender to the mail clients of the recipient. Messages can be filtered by the incoming mail server (in green) or by an online mail client (in blue).

Mail filtering language (Sieve)

Sieve is a language for filtering messages on the incoming mail server. It is specified in RFC 5228 and it is fairly simple: Using the control commands `if`, `elsif`, and `else`, you can specify under which conditions a specific action shall be applied. You can find plenty of examples throughout the RFC as well as here and here. There are just a couple of things you should know to understand them:

- **Arguments:** Most commands in the Sieve language take arguments. Mandatory arguments are determined by their position, optional arguments are identified by a colon followed by their name. Some optional arguments can take arguments themselves: `:name value`. This is similar to arguments in the command-line interface but with `:` instead of `-` before the name. When optional arguments are not provided, their default values are used instead.
- **Extensions:** The Sieve language is extensible. A script has to list the extensions which it uses at the top of its code with require.
- **Implicit keep:** Each message is stored in the inbox unless it is moved to a folder, forwarded to an address, or discarded explicitly.
- **String lists:** Wherever a list of strings is expected, such as `["To", "Cc"]`, a string without brackets, such as `"To"`, can be used.
- **Prefix notation:** Commands and arguments are nested not with parentheses but by earlier tokens consuming later ones. For example, the negation of the condition exists `"Date" is not exists "Date"`. This is similar to the prefix notation.
- **Comments:** If you use `#` outside of double quotes, the incoming mail server ignores all characters including this one until the end of the line. Comments which span less or more than a line have to be enclosed in `/*` and `*/`.

- **No loops:** The Sieve language doesn't support loops. Each block is executed once or not at all.

You can generate simple filtering rules with the following tool. Make sure that the Argument makes sense for the chosen Action. Move requires the name of a folder, Forward an email address, Flag the name of a flag, and Reply the text of the reply.

Condition:	<input type="text" value="Subject"/>	Negation:	<input type="checkbox"/>
Address part:	<input type="text" value="Whole address"/>	Action:	<input type="text" value="Flag"/>
Match type:	<input type="text" value="Contains"/>	Argument:	<input type="text" value="\Seen"/>
Value:	<input type="text" value="Test"/>	<input type="button" value="↶"/> <input type="button" value="🗑️"/> <input type="button" value="↷"/>	

```
require "imap4flags";
if header :contains "Subject" "Test" {
    addflag "\\Seen";
}
```

Users don't have to learn the Sieve language. Mail clients can offer a graphical user interface (GUI) similar to the tool above, where users don't have to see the generated code. You find a list of all the extensions to the Sieve mail filtering language on Wikipedia.

▼ Out-of-office replies

Among the reasons for sending an automatic response to the sender of a message are:

- **Vacation notice:** Inform the sender that the message won't be read in the coming days.
- **Change-of-address notice:** Inform the sender that the recipient's email address has changed.

Prior to IMAP, where servers can support the configuration of vacation responses, Sieve and ManageSieve with the vacation extension were the only standardized way to configure such responses. According to RFC 3834, the same response should be sent to the same sender only once within a period of several days even when the sender sends additional messages.

```
require ["date", "relational", "vacation"];
if allof (currentdate :value "ge" "date" "2021-05-14", currentdate :value "le" "date" "2021-05-21") {
    vacation "Hi, I had to take a couple of days off to read ef1p.com/email. I will be back soon.";
}
```

A simple Sieve script for an automatic vacation response, which I've adapted from Gandi.

▼ Support by email service providers and mail clients

Unfortunately, none of the big free email service providers support Sieve. If you pay for your mailbox, though, chances are that you can use the Sieve language since it is implemented by the most popular mail servers. Providers with Sieve support include Fastmail, mailbox.org, ProtonMail, and Gandi. Other email service providers support server-side filters with proprietary rules through their web interface. One example is Gmail:

When a message is an exact match for your search criteria:

☐ Skip the Inbox (Archive it)
 ☐ Mark as read
 ☐ Star it
 ☐ Apply the label: Choose label...
 ☐ Forward it [Add forwarding address](#)
☐ Delete it
 ☐ Never send it to Spam
 ☐ Always mark it as important
 ☐ Never mark it as important
 ☐ Categorize as: Choose category...
 ☐ Also apply filter to 2 matching conversations.

Learn more
 [Create filter](#)

Gmail users can go to the [Filters and Blocked Addresses](#) tab of their settings and click on “Create a new filter”. While Gmail has an [API for managing filters](#), other mail clients won’t support such a proprietary protocol.

You might struggle more to find a [suitable mail client](#). When it comes to desktop clients, there’s basically just a [plugin](#) for [Thunderbird](#). If you’re willing to use a [web client](#), [Roundcube](#) has you covered as well.

Filter management protocol (ManageSieve)

ManageSieve is a protocol for managing [Sieve scripts](#) remotely. It is specified in [RFC 5804](#) and works similar to the protocols we have seen so far. After an initial greeting from the server, the client sends commands to which the server responds. Just like [IMAP](#), responses are completed with a line which starts with OK or NO; but unlike IMAP, the commands are not preceded with a tag. Just like IMAP, multiline strings are prefixed with their length; but unlike IMAP, the client can [include a plus](#) to continue with the string without having to wait for a continuation response from the server. Just like [SMTP for Relay](#), there’s no variant of ManageSieve which can be used with [Implicit TLS](#). The server sends its [capabilities](#) automatically in its greeting and after successful [STARTTLS](#) and [AUTHENTICATE](#) commands. As part of the capabilities, the server indicates which extensions to the Sieve language and which [SASL mechanisms](#) it supports. According to [RFC 5804](#), ManageSieve servers have to support [PLAIN](#) over TLS and [SCRAM-SHA-1](#).

The following tool shows you how to use the [ManageSieve commands](#) from your [command-line interface](#). Unlike the previous tools, you have to configure the address and the [port number](#) of the server manually as this information is not included in [Thunderbird’s configuration files](#). The standard describes how to locate the ManageSieve server [with SRV records](#) and the [autoconfiguration tool](#) above does query the `_sieve._tcp` subdomain. However, since virtually no one configures such SRV records (at least not for the ManageSieve protocol), I didn’t bother to implement this discovery mechanism here. ManageSieve servers listen [on port 4190](#) by default. The [Thunderbird plugin](#), which I mentioned earlier, simply [probes this port](#) on the [IMAP server](#) in order to configure itself.

Important: Since [LibreSSL](#) doesn’t support the ManageSieve STARTTLS command, you have to use [OpenSSL](#) (see the [boxes below](#)).

OpenSSL:	<input type="text" value="openssl"/>	Action:	<input type="text" value="Upload"/>
Server:	<input type="text" value="sieve.example.org"/>	Name:	<input type="text" value="MyScript"/>
Port:	<input type="text" value="4190"/>	Script:	<input data-bbox="935 1783 1297 1917" type="text" value='require "body";
if body :contains "Test" {
 discard;
}'/>
Username:	<input type="text" value="alice@example.org"/>		
Password:	<input type="password" value="Your password"/>		
List:	<input type="checkbox"/>		<input type="button" value="↶"/> <input type="button" value="🗑"/> <input type="button" value="↷"/>

```
$ openssl s_client -quiet -crlf -starttls sieve -connect sieve.example.org:4190
"IMPLEMENTATION" "{NameAndVersion}"
"NOTIFY" "{Methods}"
```

```

"SASL" "PLAIN"
"SIEVE" "{Extensions}"
"VERSION" "1.0"
OK "STARTTLS completed."
AUTHENTICATE "PLAIN" "AGFsawNLQGV4YW1wbGUub3JnAA=="
OK "AUTHENTICATE completed."
CHECKSCRIPT {62+}
require "body";
if body :contains "Test" {
    discard;
}

OK "CHECKSCRIPT completed."
PUTSCRIPT "MyScript" {62+}
require "body";
if body :contains "Test" {
    discard;
}

OK "PUTSCRIPT completed."
SETACTIVE "MyScript"
OK "SETACTIVE completed."
LOGOUT
OK "LOGOUT completed."

```

Explanation: While you can have multiple scripts on the server, at most one of them can be active. You cannot delete the active script. You can deactivate the active script by activating another script or by using an empty script name to set no script active. You can also generate the argument to PLAIN yourself with `echo -ne '\0000username\0000password' | openssl base64`.

▼ LibreSSL doesn't support ManageSieve

With the `-starttls` option, you tell `openssl` for which protocol you want to start TLS. There are two implementations of openssl: OpenSSL supports ManageSieve, LibreSSL doesn't. If you provide `-starttls sieve`, OpenSSL executes this code. Can't we use one of the other protocol options to let LibreSSL send STARTTLS to the server? The answer is no, unfortunately:

- IMAP: LibreSSL first issues `. CAPABILITY` to check whether the server supports STARTTLS. ManageSieve servers ignore this as an invalid command. LibreSSL then tries to initiate TLS anyway and sends `. STARTTLS`. Since the ManageSieve protocol doesn't use tags, this line fails to achieve what we want.
- POP3: Using `-starttls pop3` doesn't work because POP3 clients use TLS instead of STARTTLS to upgrade the connection.
- SMTP: Using `-starttls smtp` could work but for some reason it also doesn't work. LibreSSL first sends the EHL0 command, which is ignored by ManageSieve servers as an invalid command. Continuing anyway, LibreSSL sends STARTTLS to the server and doesn't check the response, which is exactly what we were looking for. Unfortunately, this still fails. If you know why, please let me know.

Of all the protocols we have seen so far, not two of them initiate TLS in the same way. Thus, if you want to use the ManageSieve protocol from the command line, you have to install OpenSSL. You can check what you have with `openssl version`.

▼ How to install OpenSSL on macOS

The easiest way to install OpenSSL on macOS is with Homebrew. You can check whether Homebrew is already installed with:

```
$ brew --version
```

If this is not the case, you can install Homebrew with:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Afterwards, you can install OpenSSL with:

```
$ brew install openssl
```

By default, OpenSSL is installed in the following location without replacing the preinstalled LibreSSL:

```
$ /usr/local/opt/openssl/bin/openssl version
```




Click here to use this as the OpenSSL command in the tool above.

Format

The format of an email message is specified in [RFC 5322](#). The goal of this chapter is to make you comfortable reading raw messages.

▼ How to display the raw message

Mail clients don't display all [header fields](#) by default. Here is how you can display the raw message as it arrived in your mailbox:

- **Gmail:** Open a message, click on  in the upper right corner, then on "Show original".
- **Yahoo:** Open a message, click on  in the bottom middle, then on "View raw message".
- **Outlook:**
 - Web: Click on  in the upper right corner, then on "View" and "View message source".
 - Desktop: Double-click a message, click on the "File" menu and then select "Properties".
- **Thunderbird:**
 - Raw message: Select a message, click on the "More" button and then "View Source" (or use `⌘U`).
 - All header fields: Click on the "View" menu, then on "Headers" and "All" (or on "Normal" to go back).
- **Apple Mail:**
 - Raw message: Click on the "View" menu, then on "Message" and "Raw Source" (or use the shortcut `⌘U`).
 - [All header fields](#): Click on the "View" menu, then on "Message" and "All Headers" (or use the shortcut `⌘H`).
 - [Change preferences](#): In the "Viewing" tab of the preferences, you can configure which header fields are displayed.

File format

Since messages, including [attachments](#), are just text, they can be stored as simple [text files](#). A common [filename extension](#) for emails is `.eml`. Such files can be viewed with any [text editor](#). Desktop clients usually have an option to save a message as a file, and among Web clients, at least Gmail allows you to download a message in the `:` menu, which is located in the upper right corner.

▼ Storage format

For their own purposes, mail clients can store messages in whatever format they want. The two formats which are used by several mail clients and servers to store messages are [Mbox](#) and [Maildir](#). By default, Thunderbird uses the former but it can also be configured to use the latter. The Mbox format is specified in [RFC 4155](#). All messages are appended in their [raw format](#) to a single file. Mbox is a [text-based format](#), which means that a given string, namely `From ...`, is used to delimit the messages and that occurrences of this string in messages have to be escaped. Storing all the messages in a single file is not ideal as it might easily get corrupted if it's not properly [locked](#) while reading from and writing to it. Additionally, this format is inappropriate for [backup systems](#) that copy the complete file and not just the [differences](#) when the content of a file has changed. Thunderbird stores the messages at `~/Library/Thunderbird/Profiles/{RandomString}.default/ImapMail/{MailServer}` on macOS. If you use another operating system, you find the storage location on [this page](#). This directory contains two files for each of your mailbox folders. For example, you should have a large INBOX file and a much smaller INBOX.msf file, which is used to index the messages in the former file. (MSF stands for mail summary file.) You can use the [tail command](#) to display the specified number of lines of the last message that you've received: `tail -n 100 INBOX`. Unless you want to transfer all your messages to a new computer, you shouldn't move or modify such files as this likely causes problems for your mail client.

▼ Apple Mail storage format

Similar to [Maildir](#), Apple Mail stores each message in a separate file at `~/Library/Mail/`. The used format is proprietary and there's no official documentation about it but it's fairly easy to [reverse engineer](#). After a folder with the version number of the format, `V7` in my case, Apple Mail generates a folder for each of the added email accounts with a [Universally Unique Identifier \(UUID\)](#) as its name. Inside these accounts folders, Apple Mail generates a folder ending with `.mbox` for each of the [IMAP folders](#), such as `INBOX.mbox`, `Sent Messages.mbox`, and so on. These mailbox folders contain another folder with a UUID, which finally contains the Data folder with the actual messages in further folders. Put together, the folder nesting is as follows: `~/Library/Mail/V7/{UUID}/INBOX.mbox/{UUID}/Data`.

Apple Mail enumerates the messages with a single counter across all your accounts. It uses the [filename extensions](#) `.emlx` for messages without attachments and `.partial.emlx` for messages with attachments. In these `emlx` files, Apple Mail prepends the length of the message in bytes to the raw message and appends a [property list](#) with additional information. It's a text-based format that you can open with any [text editor](#). The messages are stored in a Messages folder inside the Data folder with their number used as their name. For example, you might

have .../Data/Messages/123.emlx. If the message contains attachments, Apple Mail removes the attachments (at least in most cases) and stores them separately in an Attachments folder. For example, if message 123 has an attachment, the message is stored at .../Data/Messages/123.partial.emlx and its attachment at .../Data/Attachments/123/{Position}/Filename.pdf. The Position encodes where the attachment was included in the message's multipart hierarchy. In an effort to limit the number of files to 1'000 per folder, Apple Mail creates subfolders when the message number becomes larger than 999. For example, message 1234 is stored at .../Data/1/Messages/1234.emlx, message 12345 at .../Data/2/1/Messages/12345.emlx, and message 123456 at .../Data/3/2/1/Messages/123456.emlx.

Please note that you have to give the Terminal full disk access in the "System Preferences" under "Security & Privacy" and then "Privacy" if you want to access the ~/Library/Mail/ folder from the command line because of the System Integrity Protection (SIP) of macOS. With full disk access enabled, you can find the message with a particular number with `find ~/Library/Mail/ -name '1234.*emlx'`. If you need to convert .emlx files back to .eml files, for example to migrate them to a different mail client or email service provider, you may want to have a look at [this project](#).

Line-length limit

According to RFC 5322, each line of a message may consist of at most 1'000 ASCII characters, including CR+LF. Implementations are free to accept longer lines, but since some implementations cannot handle longer lines, you shouldn't send them. The RFC even recommends limiting lines at 80 characters to accommodate clients that truncate longer lines in violation of the standard. In order to leave the line wrapping to the mail client of the recipient, the mail client of the sender has to encode the body if the body contains lines which are too long. If a header field is too long, it must be broken into several lines with folding whitespace: {CR}{LF} followed by at least one space or tab. If a line in the header section of a message starts with whitespace, its content belongs to the header field on the previous line. The procedure of breaking lines as done by the sender is called *folding*, the procedure of joining lines as done by the recipient is called *unfolding*. When unfolding, runs of whitespace characters are replaced with a single space character.

Message identification

There are three header fields to identify the current message and the previous messages in the same thread:

- **Message-ID:** The Message-ID identifies the current message. Its format is <{Value}@{Domain}>. Although outgoing mail servers may add this field if it's missing, the Message-ID should be chosen by the mail client. Otherwise, the copy stored in the sent folder on the incoming mail server lacks this field, which defeats its purpose. Whoever chooses the Message-ID should make sure that it's unique. Mail clients often choose the Value as a universally unique identifier (UUID) and the Domain as the domain part of the user's email address. The sender has to decide whether two messages are the same and thus share the same Message-ID. If the client generates different versions of the same message due to BCC recipients, it should use the same Message-ID for all of them.
- **In-Reply-To:** If a user replies to a message, the Message-ID of the replied-to message is put into the In-Reply-To header field.
- **References:** While In-Reply-To refers only to the direct parent message, the References field lists the Message-IDs of all ancestor messages, including the direct parent message. This is useful to reconstruct a conversation even if not all intermediary messages were sent to you. Clients compose this field by adding the Message-ID of the replied-to message to the References of the replied-to message. When determining which messages belong to the same thread, clients use additional heuristics, such as comparing the Subject line after stripping common prefixes, to avoid grouping messages where a person replies to a message just to send an unrelated message to the sender of the message.

```
Message-ID: <64F0D157-FD89-4858-9589-9BDD22870B22@example.org>
In-Reply-To: <BDC9DF60-660C-435D-B909-C16567A8470C@example.com>
References: <BF5BBB3B-92F9-42E8-9353-A7579CF53E45@example.net>
            <BDC9DF60-660C-435D-B909-C16567A8470C@example.com>
```

An example of what the three message identification header fields look like.
The References field contains the message ID of the In-Reply-To field.

▼ Mandatory header fields

According to RFC 5322, only two header fields must be included in every message: the From field and the Date field. While not strictly mandatory, every message should have a Message-ID, and every reply should have an In-Reply-To and a References header field if the replied-to message had a Message-ID.

▼ Quoting the previous message

It's a common practice to quote the text of the original message in the reply, but this is completely optional, and the format for doing so isn't standardized. Most mail clients prefix quoted lines with the greater-than sign in a text-based response and wrap the quoted text in a `<blockquote>` element when using HTML. Modern clients typically display quoted text with a vertical bar.

```
Text of the current message
> First line of the parent message
> Second line of the parent message
>> Quoted text in the parent message
```

How text is usually quoted at different nesting levels. Quoting the text allows you to reply below each paragraph of the original message.

If mail clients quote the message to which you reply, they also add an attribution line, which mentions the author and the date of the original message. Quoting text with > is mentioned only in [RFC 1849](#) and [RFC 3676](#), the former being related to [Usenet](#) rather than traditional email. One problem of quoting text with > is that this can cause lines to exceed the imposed length limit.

▼ Universally Unique Identifier (UUID)

Universally Unique Identifier (UUID) is a standard for generating globally unique identifiers without coordination among the involved parties. The standard has been published by various organizations, including IETF in [RFC 4122](#). A universally unique identifier is a 128-bit number, which is encoded as 32 hexadecimal digits with 4 hyphens inserted at fixed positions. The format of UUIDs is XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX, where the four bits of A encode the used algorithm, and the first one to three bits of F encode the used format. Please note that I use A and F as variable names here. All the bits, including the actual values of A and F, are encoded as hexadecimal digits. X stands for four bits. How those bits are determined depends on A and F.

The format F is in binary either 0xxx for the original format, 10xx for the RFC format, or 110x for Microsoft's format. 111x is reserved for a potential future format, and the lowercase x stands for a single bit.

When using the RFC format, the algorithm A, which is used to determine the remaining bits, is one of the following:

- 1: The remaining bits consist of the current timestamp and the MAC address of the device which generated the UUID.
- 2: A variant of algorithm 1 used in the Distributed Computing Environment (DCE) by the Open Software Foundation (OSF).
- 3: The MD5 hash of a namespace identifier and a name within that namespace. A and F overwrite six bits of the MD5 hash.
- 4: The remaining 122 bits are chosen randomly. The message IDs in the example above were chosen with this algorithm.
- 5: Algorithm 5 is the same as algorithm 3 but it uses SHA-1 instead of MD5 as the cryptographic hash function.

Trace information

According to [RFC 5321](#), whenever a mail server receives a message, it must add a Received header field at the beginning of the message without changing or deleting already existing Received header fields. Received header fields have the following format:

```
Received: from {EhloArgument} ({DnsReverseLookup} {IpAddressOfClient})
  by {DomainNameOfServer}
  with {Protocol}
  id {SessionId}
  for {AddressOfRecipient};
  {DayOfWeek}, {Day} {Month} {Year} {Hour}:{Minute}:{Second} {TimeZone}
```

The format of Received header fields. The curly brackets stand for values which need to be inserted. The with, id, and for clauses are optional. The newlines can be in other places, and additional information is often added as comments in parentheses in various places.

According to [RFC 5321](#), the Protocol is either SMTP or ESMTP. [RFC 3848](#) specified additional values: ESMTPA when ESMTP is used with successful user authentication, ESMTPS when ESMTP is used with Implicit or Explicit TLS, and ESMTPSA when the session has been secured and the user has been authenticated. [RFC 8314](#) specifies an additional tls clause, which can be used after the for clause to record the TLS ciphersuite which has been used. Gmail adds such information as a comment instead: (version=TLS1_2 cipher=ECDHE-ECDSA-CHACHA20-POLY1305 bits=256/256). Checking the Received header fields of a received message gives you an idea whether the message was secured during transport. Note, however, that Received header fields are not authenticated: The mail servers through which a message passes can change the Received header fields that were added by mail servers through which the message already passed. In addition, not all mail servers might support the newer protocol values, and relays over a private network are often not protected with TLS. A message typically has at least four Received header fields, which only makes sense if you look at the official architecture instead of the simplified architecture. A Received header field is added by the mail submission agent (MSA), the outgoing mail transfer agent (MTA), the incoming mail transfer agent (MTA), and the mail delivery agent (MDA). Here is a Received header field, which was added by my outgoing mail server:

```
Received: from [192.168.1.2] (unknown [203.0.113.167])
  (Authenticated sender: kaspar@ef1p.com)
  by relay12.mail.gandi.net (Postfix) with ESMTPSA id 7974D200009
  for <contact@ef1p.com>; Thu, 3 Dec 2020 14:14:48 +0000 (UTC)
```

What an actual Received header field looks like. I've only replaced my IP address with an [address reserved for documentation](#). Due to [Network Address Translation \(NAT\)](#), the [private IP address](#) that my computer used in the [EHLO command](#) and the public IP address that the outgoing mail server saw were different. Since my public IP address doesn't have a [reverse DNS entry](#), the server recorded the name of the client as unknown. (Authenticated sender: kaspar@ef1p.com) is a comment, which the server added to indicate which user submitted the message. (Postfix) is also a comment, indicating the name of the server implementation. Everything else matches the format which I've described [above](#).

An incoming mail server which delivers a message must add the MAIL FROM address of the [envelope](#) in a Return-Path header field to the message. While a message can have several Received header fields, it may have at most one Return-Path header field. If a message is resubmitted, for example by a [filtering rule](#), the Return-Path header field should be removed, and its value should be used as the MAIL FROM address. As we discussed [earlier](#), the Return-Path header field can be different from the From header field.

```
Return-Path: <kaspar@ef1p.com>
```

What a Return-Path header field looks like.

▼ Recover why you received a message

Since the Bcc recipients are [usually removed from the message](#) even for the Bcc recipients themselves, mail clients don't know whether a message has been forwarded or whether the user was a hidden recipient if the user's address is not listed among the recipients. By inspecting the Received header fields, mail clients could easily distinguish between the two scenarios in most cases: If the message has been forwarded, there should be a Received header field with one of the recipient addresses in the for clause. Recovering the address through which a message has been forwarded to your mailbox could be useful for [filtering](#) incoming messages into different folders automatically. And as we discussed [earlier](#), mail clients shouldn't offer a reply-to-all option for messages where the user was a Bcc recipient as this would leak what the sender tried to hide by using the Bcc field.

▼ Local Mail Transfer Protocol (LMTP)

The [Local Mail Transfer Protocol \(LMTP\)](#) is a variant of the [Extended Simple Mail Transfer Protocol \(ESMTP\)](#), in which the server can reject an incoming message for each recipient individually. In the case of ESMTP, the server can send only a single reply after the message has been transferred. If the message can be delivered to some of the recipients but not all of them, the ESMTP server has to queue the message in order to deliver it to the pending recipients at some later point. In the case of LMTP, the server has to confirm the acceptance of the message for each recipient which was provided with the RCPT TO command. Being able to reject a message for individual recipients frees LMTP servers from having to manage a mail queue.

LMTP is specified in [RFC 2033](#) and may be used only in a local network. LMTP uses LHL0 instead of EHLO to greet the server. The reason why I mention LMTP is because you might encounter it as LMTP [S] [A] in the with clause of a Received header field. LMTP also pops up in other places, for example in the [code](#) to which I linked [earlier](#).

Content encoding

[RFC 5322](#) specifies a format for text messages, whose lines may consist of at most [1'000 ASCII characters](#). Whenever the content of a message doesn't fulfill this requirement, it must be encoded according to the [Multipurpose Internet Mail Extensions \(MIME\)](#), as specified in [RFC 2045](#). When mail clients encode messages according to MIME, they indicate this with the following header field:




```
MIME-Version: 1.0
```

The header field used to indicate that a message is formatted using MIME.




In theory, the version number allows the Internet community to make changes to the standard. In practice, however, the standard [didn't specify](#) how mail clients are supposed to handle messages with an unknown MIME version. As a consequence, you cannot change the version number without breaking email communications, which makes this header field completely useless. The version 1.0 survived the last 30 years and will likely survive the next 30 years. MIME also introduced additional message header fields, which we'll cover in this and the following subsections.

Unless all involved SMTP servers support the BINARYMIME extension as specified in [RFC 3030](#), which is rarely the case, content containing non-ASCII characters or lines longer than 1'000 characters must be encoded with one of the following two methods:

- **Quoted-Printable:** Any byte which doesn't represent a printable ASCII character is encoded with the equality sign followed by the value of the byte encoded as two hexadecimal digits. Since = is used as the escape character, it has to be encoded with its hexadecimal ASCII value as =3D. Lines may be at most 78 characters long, including {CR}{LF}. Longer lines have to be broken by inserting = {CR}{LF}. All sequences of these three characters are removed when decoding the Quoted-Printable encoding. Since some mail servers add or remove trailing whitespace, tabs and spaces which are followed by {CR}{LF} also need to be encoded with hexadecimal digits. Any sequence of bytes can be encoded with this method. The Quoted-Printable encoding only makes sense, though, if most of the bytes are printable ASCII characters. This is the case for those European languages which share most of their characters with the English alphabet. Texts in such languages remain largely readable when using the Quoted-Printable encoding. The probability that a random byte falls into the range of printable ASCII characters is just a bit bigger than one third, though. Thus, the size of binary data, such as images, more than doubles with this encoding. The following tool allows you to encode and decode Quoted-Printable:

Decoded: Encoded: Charset:   

- **Base64:** Binary data and non-Western-European languages are best encoded with Base64. While hexadecimal digits encode 4 bits each, Base64 digits encode 6 bits each. 6 bits can represent $2^6 = 64$ different values. Base64 uses the characters A – Z, a – z, 0 – 9, +, and / to encode these 64 values. What makes the Base64 encoding special is that bytes and digits don't align: Three bytes are encoded with four Base64 digits. If you shift the input by one or two bytes, the Base64 encoding looks completely different. If the size of the input is not a multiple of three, one or two equality signs are appended to the output in order to make the output a multiple of four. This procedure is known as padding. In order to respect the line-length limit, a line break is inserted after at most 76 Base64 characters. Base64 encoding increases the size of the content by 33% and the line breaks add another 2.6% on top of that. You can encode and decode Base64 with the following tool:

Decoded: Encoded: Charset:   

The mail client of the sender informs the mail client of the recipient with the following header field that the content is encoded:

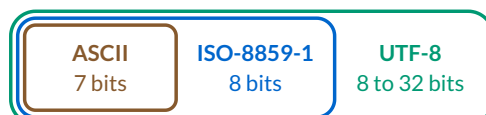
Content-Transfer-Encoding: {Value}

This header field indicates with which method the content of the message has to be decoded.
The Value can be quoted-printable, base64, or 7bit if no content encoding has been used.
(If the 8BITIME or BINARYMIME extensions are supported, the value can also be 8bit or binary.)

If the message already consists of only printable ASCII characters, the line-length limit can also be achieved with soft line breaks.

▼ Character encoding (charset)

The character encoding determines how each character is encoded as a sequence of zeros and ones. I've already covered this in the previous article. What we're interested in now is how this affects the Quoted-Printable and Base64 encodings. The most popular character encodings are ASCII, ISO-8859-1, and UTF-8. ASCII encodes each character with 7 bits, leaving the 8th bit in each byte unused. Its set of characters (charset for short) includes only the English alphabet. ISO-8859-1 extends ASCII with characters used in Western European languages, such as à, á, â, ã, ä, å, and the like. Each ISO-8859-1 character is encoded with 8 bits. UTF-8, on the other hand, encodes all the code points defined by Unicode with 1, 2, 3, or 4 bytes. Both ISO-8859-1 and UTF-8 encode the ASCII characters if the first bit of a byte is zero. For non-ASCII characters, UTF-8 needs at least two bytes. Therefore, if all the characters in a message can be encoded with ISO-8859-1, the Quoted-Printable and the Base64 encodings are shorter if the input string is encoded with ISO-8859-1 rather than with UTF-8. You can verify this with the tool above: The Quoted-Printable encoding of the inverted exclamation mark is =A1 when using ISO-8859-1 and =C2=A1 when using UTF-8.



ASCII encodes only a subset of the characters defined in ISO-8859-1, and
ISO-8859-1 encodes only a fraction of the characters available in UTF-8.

▼ Percent encoding (URL encoding)

If you ever did some [web development](#), you might have encountered the [Percent encoding](#). It's used to encode arbitrary data in a [Uniform Resource Identifier \(URI\)](#), such as a [Uniform Resource Locator \(URL\)](#). The Percent encoding is specified in [RFC 3986](#), and it works similar to the Quoted-Printable encoding. The difference is that the Percent encoding has a longer list of [reserved characters](#) and that the [percent sign](#) is used as the escape character instead of the equality sign. Additionally, [whitespace](#), including newlines, have to be encoded, and Percent encoding is usually used on UTF-8 strings. I've included this box and the following tool mostly for the sake of completeness. The only place where you might find Percent-encoded strings in emails is in [links](#) of [HTML messages](#).

Decoded:

Form: ☐

Strict: ☐



Encoded:

▼ Decoding on the command line

If you use [POP3](#) or [IMAP](#) to fetch messages from your [command-line interface](#), you likely also want to decode the received messages on the command line. The following commands read the string to encode or decode from their [standard input](#) and write the encoded or decoded string to their [standard output](#). This allows you to use the commands both in [pipelines](#), such as `echo -n 'input' | {Command}`, and with [files](#), such as `{Command} < input.txt > output.txt`.

```
$ qprint -e
$ qprint -d
```

How to encode (-e) and decode (-d) Quoted-Printable with [qprint](#). Use `brew install qprint` to install this command on macOS.

```
# You have to install the package only once:
$ npm install -g quoted-printable
$ quoted-printable -e
$ quoted-printable -d
```

How to encode (-e) and decode (-d) Quoted-Printable with the [quoted-printable](#) package if you already have [Node.js](#).

```
$ openssl base64 -e
$ openssl base64 -d
```

How to encode (-e) and decode (-d) Base64 with [OpenSSL](#). Use the option -A to have no newline characters inserted or expected.

```
$ perl -MMIME::QuotedPrint -0777 -nle 'print encode_qp($_)
$ perl -MMIME::QuotedPrint -0777 -nle 'print decode_qp($_)

$ perl -MMIME::Base64 -0777 -ne 'print encode_base64($_)
$ perl -MMIME::Base64 -0777 -ne 'print decode_base64($_)

$ perl -MURI::Escape -0777 -ne 'print uri_escape($_)
$ perl -MURI::Escape -0777 -ne 'print uri_unescape($_)'
```

How to encode and decode Quoted-Printable, Base64, and Percent with [Perl](#), which is likely preinstalled on your computer. You can use [explainshell.com](#) to learn more about the used options. The code uses the [MIME::QuotedPrint](#), [MIME::Base64](#), and [URI::Escape](#) modules.

```
$ python3 -c 'import sys, quopri; sys.stdout.buffer.write(quopri.encodestring(sys.stdin.buffer.read()))'
$ python3 -c 'import sys, quopri; sys.stdout.buffer.write(quopri.decodestring(sys.stdin.buffer.read()))'

$ python3 -c 'import sys, base64; sys.stdout.buffer.write(base64.b64encode(sys.stdin.buffer.read()))'
$ python3 -c 'import sys, base64; sys.stdout.buffer.write(base64.b64decode(sys.stdin.buffer.read()))'

$ python3 -c 'import sys, urllib.parse; print(urllib.parse.quote(sys.stdin.read()))'
$ python3 -c 'import sys, urllib.parse; print(urllib.parse.unquote(sys.stdin.read()))'
```

How to encode and decode Quoted-Printable, Base64, and Percent with [Python](#), which is likely preinstalled on your computer. The commands use the [quopri](#), [base64](#), and [urllib.parse](#) modules and the first four commands operate on the raw bytes.

Header encoding

[RFC 2047](#) specifies how one can use non-ASCII characters in [certain header field values](#), such as the [subject](#) and the [display names](#). Instead of introducing new header fields to specify the encoding of existing header fields, encodings in header fields indicate which [character encoding](#) and which [content encoding](#) has been used. This results in the so-called [Encoded-Word encoding](#). Its format is as follows: `=?{CharacterEncoding}?{ContentEncoding}?{EncodedText}?=`, where `CharacterEncoding` is usually either ISO-8859-1 or UTF-8, `ContentEncoding` is either Q for Quoted-Printable or B for Base64, and `EncodedText` is the field value encoded according to the previous parameters. The Quoted-Printable encoding is slightly modified when used to encode header field values: Question marks, tabs, and underlines are escaped with their hexadecimal representation and spaces are encoded with underlines. In order to adhere to the [line-length limit](#), whitespace between adjacent Encoded Words is removed completely, which allows the encoder to break long words with a newline (and also to mix different character encodings). The following tool does all of that for you. It uses Quoted-Printable or Base64 depending on which encoding is shorter, and it supports only ISO-8859-1 and UTF-8.

Decoded: ¡Buenos días!

Encoded: =?ISO-8859-1?Q?=A1Buenos_d=EDas!?=



In case you haven't noticed yet: The [ESMTP tool](#) above automatically encodes the Subject and the Body if necessary. If you want to use non-ASCII characters in display names, you have to paste the Encoded Word into the address field yourself. The following boxes explain how non-ASCII characters are supported in [domain names](#), which is really interesting but also fairly advanced.

▼ Punycode encoding

[Punycode](#) is yet another encoding of [Unicode](#) with [ASCII characters](#). While domain names may consist of arbitrary bytes, many protocols require that the domain names of servers contain only letters, digits, and hyphens (LDH) from the ASCII character set. This is known as the [preferred name syntax](#), and (E)SMTP is one of the protocols which [uphold the LDH rule](#). In order to remain backward compatible and to require no changes to the [DNS infrastructure](#), domain names with non-ASCII characters have to be encoded with just ASCII letters, digits, and hyphens. Punycode is an encoding which does exactly that. It is specified in [RFC 3492](#) and it tries to be as space-efficient as possible.

Punycode encodes Unicode strings in three steps:

- 1. Remove and sort the non-ASCII characters:** In the first step, the Punycode encoder removes all non-ASCII characters from the string which is to be encoded. For example, Zürich, the city in which I live, becomes Zrich. Since ü is the only non-ASCII character, there is nothing to sort. If there were several non-ASCII characters, the encoder would have to sort them according to their Unicode [code point](#).
- 2. Determine the deltas:** In the second step, the encoder determines how many iterations the decoder has to do nothing before inserting the non-ASCII characters back in. The decoder loops through the positions of the current string and through all Unicode characters. In the first iteration, the decoder would add the first non-ASCII code point at the first position. Since ASCII uses all 7 bit numbers from 0 to 127, the first non-ASCII code point is 128. In the second iteration, the decoder would add the character with the code point 128 at the second position, and so on. Once the decoder reaches the last position of the string, it goes back to the first position to potentially insert the next higher code point there. Let's look at our example again. There are six positions where a character might be inserted: 122r3i4c5h6. The first (and only) character to insert is ü with the [code point 252](#). The decoder has to loop through the string $252 - 128 = 124$ many times before it is ready to insert the character ü. Since the string has six positions and we want to insert the ü at the second position, the decoder has to do nothing for $124 \times 6 + 1 = 745$ iterations before inserting the current character ü at the current position 2. If there were more characters to insert, there would now be seven positions for doing so. The decoder would continue from its current state (ü at position 2) and skip again the number of iterations as specified by the encoder. The number of skipped iterations is called "delta". The result of this second step is a delta for each non-ASCII character which needs to be inserted into the string of ASCII characters as determined by step one. While Zrich [745] decodes to Zürich, Zrich [745, 0] decodes to Züürich, Zrich [745, 1] decodes to Zürüich, Zrich [745, 2] decodes to Züriüich, and so on.
- 3. Encode the deltas:** In the third step, the encoder encodes the list of numbers from the second step with letters and digits. A hyphen is used to separate the encoded deltas from the string of ASCII characters from step one. To make the encodings as compact as possible, Punycode encodes the deltas without a delimiter between them. It uses [variable-base integers](#) with a variable termination threshold instead. Since domain names in the preferred name syntax are case-insensitive, the case of the letters may not matter for the encoding of the deltas. The letters a to z represent the decimal numbers 0 to 25 and the digits 0 to 9 represent the decimal numbers 26 to 35. Unlike all the numbers you're used to, the positions of Punycode numbers get more significant to the right. Each position has its own threshold and its own base. If a digit at a position is below the threshold there, it marks the end of the current number. Let's imagine, for a moment, that we use only the digits 0 to 9 and a fixed threshold value of 5. Counting then works as follows: 0, 1, 2, 3, 4 (so far, each number has been terminated by the digit being below the threshold, but from now on we need an additional digit to terminate the number), 50, 60, 70, 80, 90 (we cannot go back to a digit below 5 in the first position as this would terminate the number so we choose the base of the second position to be 10 minus the threshold of the first position), 51 ($5 \times 1 + 1 \times 5 = 10$), 61 ($6 + 5 = 11$), 71, 81, 91 ($9 + 5 = 14$), 52 ($5 \times 1 + 2 \times 5 = 15$), and so on. After 94 comes 550 and after 990 ($9 \times 1 + 9 \times 5 = 54$) comes 551 ($5 \times 1 + 5 \times 5 + 1 \times 25 = 55$). The base in the third position is

determined by multiplying the base in the second position with the number of available symbols minus the threshold in the second position ($5 \times (10 - 5) = 25$). The base in the fourth position will be $25 \times (10 - 5) = 125$, and so on. The higher the threshold value, the more likely it is that you don't need an additional digit to terminate the number. On the other hand, a higher threshold value means that the base at the next position is lower. This in turn means that less progress is made in the next position and an additional position might be needed. Punycode sets the threshold as the position times the number of symbols minus the current bias and limits all thresholds to a certain range. The bias is 72 initially and the range for thresholds is 1 to 26. Thus, the threshold at position 1 is $\max(1, 1 \times 36 - 72) = 1$, the threshold at position 2 is $\max(1, 2 \times 36 - 72) = 1$, and the threshold at position 3 is $\min(26, 3 \times 36 - 72) = 26$ initially. The bias is adapted after each delta because the current delta indicates the likely size of the next delta. In our example, 745 is encoded as kva. Since k stands for 10, v for 21, and a for 0, $10 \times 1 + 21 \times 36 + 0 \times (36 \times 36)$ indeed equals 745. Since the threshold is always at least 1, a always terminates the current delta. While Zrich-kva decodes to Zürich, Zrich-kvaa decodes to Züürich, Zrich-kvab decodes to Zürüich, Zrich-kvac decodes to Züriüich, and so on. The bias is adapted to 0 after the first delta, which makes the threshold at the first position $\min(26, 1 \times 36 - 0) = 26$. This means that Zrich-kvaz is a valid encoding while Zrich-kva0 is not because the 0 (representing 26) needs to be terminated with an a: Zrich-kva0a. You can try all of this yourself with the following tool. The domain name option is explained in [another box](#).

Decoded: Encoded: Domain: ☐   

Warning: The domain option is a very crude approximation of the standard. Use the [official utility](#) when correctness matters!

A few additional observations:

- Punycode transforms a sequence of Unicode code points irrespective of their encoding, such as UTF-8 or UTF-16.
- The deltas can be only positive. This is why the non-ASCII characters have to be sorted before they can be encoded.
- If the encoded word contains a hyphen, then the decoded word contains ASCII characters and the last hyphen is interpreted as the delimiter between the ASCII characters and the deltas. If the decoded word doesn't contain ASCII characters, then the encoded word doesn't contain a hyphen. a is encoded as a-, - as --, ü as tda, and the empty string as the empty string.
- Punycode encodes non-ASCII symbols like i and æ with letters, digits, and hyphens, but it doesn't escape the remaining printable ASCII characters, such as !, =, and &. Punycode would be more flexible if the initial state started with a code point of 0 instead of 128. As we will see soon, this doesn't matter for internationalized domain names, though.
- After a potentially large initial delta, the subsequent deltas are small if all the characters come from the same language. This is what makes Punycode so efficient. For example, Ελληνικά is encoded as twa0c6aifdar, which consists of just four more characters. Even more astonishingly, the UTF-8 encoding of Ελληνικά takes 16 bytes, whereas the UTF-8/ASCII encoding of twa0c6aifdar takes just 12 bytes.

▼ Unicode normalization

Unicode is designed to be as inclusive as possible. Any character and symbol that people want to express gets included in the standard. While a unified encoding of all writing systems and earlier character encodings is great for interoperability, it's really bad for comparing strings because characters that we humans consider to be equal can be encoded by different code points. When you search for a string encoded in one variant, you also want to find strings encoded in other variants. For this reason, Unicode strings need to be normalized before comparing them so that identical strings have the same binary representation.

Unicode normalization distinguishes between encodings that are syntactically identical and encodings that are semantically similar but not identical. The former is called canonical equivalence, the latter compatibility equivalence. Additionally, some characters can be represented by a single code point or by several code points. The former is the composed representation, the latter the decomposed representation. Based on these options, Unicode defines the following four normalization forms (NF):

	Composition	Decomposition
Canonical	NFC	NFD
Compatibility	NFKC	NFKD




The four normalization forms of Unicode.

Replacing characters by compatibility equivalence also replaces characters that are canonically equivalent. There are no normalization forms for the latter without the former. The relationship between canonical equivalence and compatibility equivalence can thus be visualized as follows:



Compatibility equivalence includes canonical equivalence.

Before we look at examples, let me introduce the following tool to you. It outputs the code points of the given input after applying the given normalization. It uses [JavaScript's normalize function](#) for the Unicode normalization and it allows you to input characters by their code point(s) with [JavaScript's escape notation](#), which means that you can specify a code point with two, four, or a variable number of hexadecimal digits: `\xxx`, `\uXXXX`, and `\u{X...X}`, where X represents a hexadecimal digit.

Input: Normalization:   

Output: C a f é
43 61 66 E9

Examples of canonical equivalence:

- Combining characters such as diacritical marks: $\text{ü} \leftrightarrow \text{u} \cdot \text{ï} \leftrightarrow \text{i} \cdot \text{é} \leftrightarrow \text{e} \cdot \text{ñ} \leftrightarrow \text{n} \cdot \text{ç} \leftrightarrow \text{c} \cdot \text{æ} \leftrightarrow \text{a} \cdot \text{œ} \leftrightarrow \text{o} \cdot$
- The order of combining marks is irrelevant: $\text{a} \cdot \text{é} \cdot \text{é} \rightarrow \text{a} \cdot \text{é} \cdot \text{é} \rightarrow \text{a} \cdot \text{é} \cdot \text{é}$
- Same symbol with different semantics: Kelvin K \rightarrow Latin K, Ohm Ω \rightarrow Greek Omega Ω

Examples of compatibility equivalence:

- Style variants: $\text{ℕ} \rightarrow \text{N}$, $\text{ℕ} \rightarrow \text{N}$
- Enclosed alphanumerics: $1 \rightarrow 1$, $(1) \rightarrow (1)$, $\text{Q} \rightarrow 1$, $(a) \rightarrow (a)$, $\text{Q} \rightarrow a$
- Halfwidth and fullwidth forms: $A \rightarrow A$, $力 \rightarrow 力$
- Superscripts and subscripts: $^1 \rightarrow 1$, $1 \rightarrow 1$
- Number forms: $\frac{2}{3} \rightarrow 2/3$, $\text{IV} \rightarrow \text{IV}$
- Ligatures: $\text{ff} \rightarrow \text{ff}$, $\text{fi} \rightarrow \text{fi}$ (The ligature on the right-hand side of the second example is created by the font on this website.)
- Digraphs: $\text{ij} \rightarrow \text{ij}$, $\text{dž} \rightarrow \text{dž}$
- Letter-like symbols: $^{\circ}\text{C} \rightarrow ^{\circ}\text{C}$, $\text{‰} \rightarrow \text{‰}$, $\text{™} \rightarrow \text{™}$
- Line-breaking behavior: non-breaking space \rightarrow space, non-breaking hyphen \rightarrow hyphen (\neq hyphen-minus)

A few additional remarks:

- **Notation**: I used \leftrightarrow when the left side can be converted to the right side and vice versa, and \rightarrow when the left side is normalized to the right side and the right side can no longer be reverted to the left side.
- **Lossy conversion**: Both the canonical normalization and the compatibility normalization lose information, but in the case of canonical normalization, the loss is usually desired. In general, a normalized string cannot be reverted to its original form.
- **Idempotence**: As long as the same normalization is used, applying the normalization repeatedly doesn't change the result. More formally, $\text{normalize}(\text{normalize}(\text{Input})) = \text{normalize}(\text{Input})$. (Earlier Unicode versions had exceptions to this rule.)
- **Substring**: If a string is normalized, then so are all its substrings.
- **Concatenation**: Even if two strings are normalized, their concatenation might not be normalized.
- **Growth through NFC normalization**: In rare situations, NFC normalization can make a string longer.
- **Surprises**: While most normalizations are quite reasonable, you will get unexpected results if you play long enough with the above tool. For example, $\frac{2}{3}$ normalizes to $2/3$, but the latter uses the fraction slash and not the ASCII slash. Similarly, the hyphen doesn't normalize to the ASCII hyphen or minus. And while the trademark symbol normalizes to ™ , the copyright symbol stays the same. Depending on your requirements, you may therefore want to replace additional characters.
- **Clipboard**: It's not always clear when programs normalize strings, which can lead to subtle bugs. For example, when I copy a string to my clipboard with Firefox on macOS, the string gets normalized to NFC when pasted into other programs. I assume this is due to how Firefox stores text to the clipboard. If I copy a string with Chrome, its form is preserved, even when pasted in Firefox. This is why I write "depends on your system" next to the no normalization option in the tool above. The tool itself won't transform the string in this case, but the string might have been normalized before it reached the input field. Another example is that Chrome used to NFC normalize strings when submitting a form, which led to problems for certain languages.
- **Verification**: Since you can't be certain how programs interact with the clipboard, you have to do a hex dump if you want to verify how text has been stored to a file by a certain program: `hexdump -C file.txt`.
- **Programming**: If you copy `'mañana' === 'mañana'` to the JavaScript console of your web development tools, you get `false` because `'ma\xF1ana' !== 'ma\u00F1ana'`. If you want to prank a friend, replace the ordinary semicolon ; with the Greek question mark ; in their source code. In general, you want to store and compare NFC-normalized strings, which solves both of the just-mentioned problems. For developers, the complexity of Unicode is quite scary. Presumably simple things like counting the number of symbols in a string or reversing a string become surprisingly difficult – even before considering right-to-left (RTL) text and its combination with left-to-right (LTR) text into bidirectional (BiDi) text.
- **Invisible characters**: `'hi'.normalize('NFKC') === 'hi'.normalize('NFKC')` is `false` because the right `hi` contains a zero-width space, which is not normalized away even under compatibility equivalence.
- **Emojis**: Modifiers change the appearance of the preceding emoji. This is how skin tones, hair styles, gender, professions, and families are encoded. Click on the following emojis to see what they're made of: 🍌, 🍌👤, 🍌👤👤, 🍌👤👤👤, and 🍌👤👤👤👤. The zero-width joiner (ZWJ) is used to combine characters which also exist separately, and the variation selector 16 with the code point `FE0F` is used to render the preceding character as

an emoji rather than as a text symbol. For example, `\u26A0` gives you 🚫, whereas `\u26A0\uFE0F` gives you 🚫. Please note that such emojiification is not supported by all fonts.

- **Artistic use:** Unicode can also be used to change the appearance of ASCII text. For example, you can [flip text upside down](#) or overuse diacritics, which results in so-called [Zalgo text](#).
- **Sources:** To learn more about normalization, you can read the [technical report](#) and the [FAQ](#) by the [Unicode Consortium](#).

▼ Unicode case folding

The [domain name system](#) has been [case-insensitive](#) since its inception. This means that if you search for `EF1P.com`, you still get the records for `ef1p.com`. Furthermore, if I have a DNS record at `www.ef1p.com` and a [wildcard record](#) at `*.ef1p.com`, querying `WWW.EF1P.COM` returns the former. Since DNS servers are supposed to [preserve the case](#), they have to do the case-insensitive comparison of ASCII strings. (In theory, you're supposed to get back the domain name [as it's capitalized in the zone file of the authoritative name server](#). In practice, however, many DNS servers use case-insensitive name compression in their responses, which means that you often get back the domain name as you capitalized it in your query. Pointing from the answer section to the question section in order to make the DNS response smaller even if the case doesn't match is [explicitly allowed by the RFC](#).) Since DNS servers don't know about Punycode and Punycode encodes non-ASCII uppercase and lowercase letters differently, [internationalized domain names](#) have to be case-normalized on the client-side because users expect that case-insensitivity also applies to internationalized domain names, such as [ÖBB.at](#) and [öbb.at](#).

Unicode distinguishes between [case mapping](#) and [case folding](#). The former maps characters to their lowercase, uppercase, or [titlecase](#) equivalent, while the latter tries to "remove" the case for case-insensitive comparisons of text. If uppercase and lowercase letters had a [one-to-one correspondence](#), we could simply lowercase both strings before comparing them. Unfortunately, this doesn't work for Unicode strings even if we [NFKC-normalize](#) both of them to get rid of [ligatures](#). (The problem with ligatures is that they often exist only in lowercase, which means that `'ff'.toUpperCase() === 'FF'` and `'ff'.toUpperCase().toLowerCase() !== 'ff'`.) The two examples why lowercasing isn't enough for Unicode strings are the [German eszett ß](#) and the [Greek sigma ς](#). The former existed only in lowercase until 2017, at which point the [capital eszett Β](#) was officially adopted. While the capital eszett has already been added to Unicode with the code point [1E9E](#) in 2008, the capitalization of ß is still defined as `'ß'.toUpperCase() === 'SS'` but `'ß'.toLowerCase() === 'ß'`. Therefore, neither `x.toUpperCase().toLowerCase() === x` nor `x.toLowerCase().toUpperCase() === x` is true in general. The lowercase sigma ς is used only at the [end of words](#). Within words, σ is used. Since there is only one uppercase sigma, both `'ς'.toUpperCase() === 'Σ'` and `'σ'.toUpperCase() === 'Σ'`. And since Unicode maps the case of characters without considering their context, `'Σ'.toLowerCase() === 'σ'`. For these reasons, ß is mapped to `ss` and ς to `σ` before case-insensitive string comparisons. Since case folding is [guaranteed to be stable](#), this won't change in future [Unicode versions](#).

A few additional remarks:

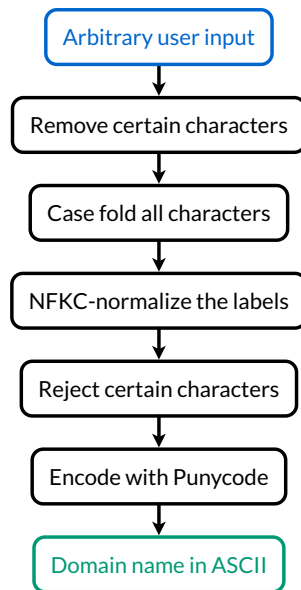
- **Duplications:** In order to keep case operations as context-independent as possible, the [Latin](#), [Greek](#), and [Cyrillic](#) scripts have [separate code points in Unicode](#) even for optically identical characters. For example, Unicode has a [Latin B](#), a [Greek B](#), and a [Cyrillic B](#), which map to `b`, `β`, and `б`. While this is great for case operations, it's [bad for internationalized domain names](#).
- **Localization:** For some characters, the case mapping still depends on the language. This is why JavaScript has a [toLocaleLowerCase](#) and a [toLocaleUpperCase](#) method. For example, in the [Turkish language](#), `'İ'.toLocaleLowerCase('tr') === 'ı'` and `'i'.toLocaleUpperCase('tr') === 'İ'`.
- **Titlecase:** [Digraphs](#), such as the [dž](#) used in Eastern European alphabets, usually exist in lowercase, uppercase, and [titlecase](#). For example, Unicode defines [dž](#), [Dž](#), and [DŽ](#). The Dutch digraph `ij`, on the other hand, is capitalized together, such as in [Jsselmeer](#), which is why only `ij` and `IJ` exist. Since digraphs are usually written as two separate characters in practice, titlecase algorithms which simply capitalize the first letter get this wrong.

▼ Internationalized domain names (IDNs)

Now that we know what [Punycode](#), [Unicode normalization](#), and [case folding](#) are, we're finally ready to discuss [internationalized domain names \(IDNs\)](#). As you might [remember](#), domain names consist of labels, which are separated by a dot. Each label of a domain name is internationalized separately. In order to distinguish Punycode-encoded labels from ordinary labels, Punycode-encoded labels are prefixed with `xn---`. This is known as the ASCII-Compatible Encoding (ACE) prefix. A label may be Punycode-encoded only if it contains non-ASCII characters. This ensures that Punycode-encoded labels never end with a hyphen. (The [preferred name syntax](#) requires that labels neither start nor end with a hyphen.) Each Punycode-encoded label may be at most 63 characters long, including the ACE prefix. If the encoding of a Unicode label is longer, the user input must be rejected.

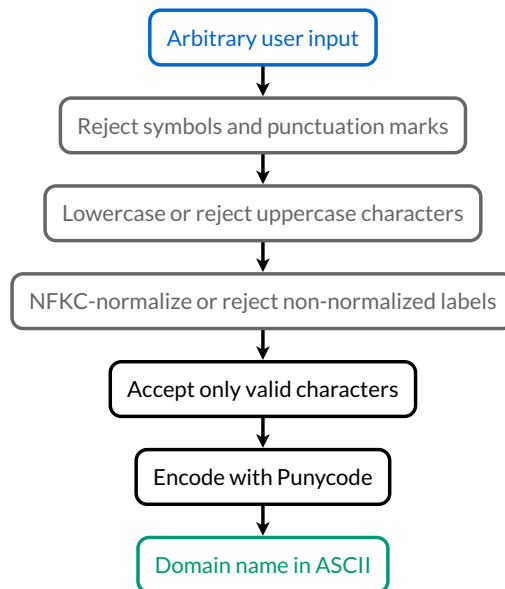
What makes internationalized domain names even more complicated is that there are two versions: IDNA2003 & IDNA2008. (IDNA stands for Internationalized Domain Names for Applications.) IDNA2008 supersedes IDNA2003, which means that IDNA2003 should no longer be used. Since a lot of the confusion comes from the differences between them, we'll look at both:

- **IDNA2003:** IDNA2003 is specified in [RFC 3490](#). It uses the [Nameprep](#) profile as specified in [RFC 3491](#) of the Stringprep algorithm, which is specified in [RFC 3454](#). Nameprep prepares an arbitrary user input to be encoded with Punycode:



How IDNA2003 normalizes user input. The normalization fails only if the output contains prohibited characters or violates the rules for bidirectional text.

- **IDNA2008:** IDNA2008 is specified in [RFC 5890](#), [RFC 5891](#), [RFC 5892](#), [RFC 5893](#), and [RFC 5894](#). Instead of prohibiting certain characters, IDNA2008 accepts only characters with specific properties, which makes it easier to migrate to newer versions of Unicode. (IDNA2003 used Unicode version 3.2 only.) Additionally, IDNA2008 no longer specifies how characters are to be mapped, it only encourages applications to meet user expectations. Removing the mapping of characters from the standard allows applications to map them according to the language which is being used. Since the IDNA standard is the same around the globe, it cannot consider the local context for character mappings.



How IDNA2008 normalizes user input. The steps in gray are required but not standardized.

So how does IDNA2008 differ from IDNA2003? Let's look at a few examples:

- **Symbols:** [P=NP.org](#) was valid under IDNA2003 but is no longer valid under IDNA2008 since symbols are no longer allowed. (Due to the limitations of Punycode, [P=NP.org](#), on the other hand, was never valid.) Disallowing symbols also prevents attackers from faking URL separators in domain names, which is a special variant of a homograph attack. For example, [ef1p.com/email.article.example](#), which uses a division slash in the domain label `com/email` under the top-level domain `.example`, was also valid under IDNA2003 but is no longer valid under IDNA2008.
- **Emojis:** Being a kind of symbol, emojis were allowed in IDNA2003 but are no longer allowed in IDNA2008. Since IDNA2003 was limited to Unicode version 3.2, only a tiny subset of emojis could be used, namely those which were originally added as text characters (mostly in Unicode version 1.1 in 1993) and given an emoji presentation in 2010. The variation selector 16 was added to Unicode in version 3.2 to render text symbols as emojis; just in time for IDNA2003. As a consequence, [❤.com](#) was once valid while [💙.com](#) never was. Emojis were intentionally disallowed in IDNA2008 because humans likely confuse different emojis even without combining characters, such as skin tones and hair styles. For example, [❤️](#) and [❤️‍🔴](#) are two different hearts, where both of them were valid under IDNA2003.

- **German eszett ß:** In IDNA2003, ß was case-folded to ss. For example, Gießen.de was transformed to giessen.de before making the DNS lookup. Since ß is allowed in IDNA2008, Gießen.de is now transformed to xn--gieen-nqa.de.
- **Greek sigma ς:** Similarly, ς was case-folded to σ in IDNA2003 but is now allowed in IDNA2008. For example, Ελλάς.gr was transformed to xn--hxa3aa7a0420a.gr in IDNA2003 and is now transformed to xn--hxa3aa3a0982a.gr in IDNA2008.

Since some characters that were previously removed, such as the zero-width joiner, are now allowed in certain contexts and other characters, such as ß and ς, are no longer mapped, some internationalized domain names are interpreted differently under IDNA2008 than under IDNA2003. These changes require a transition period from IDNA2003 to IDNA2008, where domain name registries reserve the newer mapping of an internationalized domain name for the registrant of the older mapping, bundle different mappings of a new registration, or block the registration of deviating mappings. You can read more about compatibility processing of internationalized domain names in the Unicode Technical Standard 46 and the IDN FAQ.

▼ IDNA2008 validation

Unfortunately, there is no JavaScript library to validate internationalized domain names. I've approximated the IDNA2008 rules in the Punycode tool as follows: `/^[\\p{Letter}\\p{Number}] [\\p{Letter}\\p{Mark}\\p{Number}\\p{Join_Control}]* (? : -+ [\\p{Letter}\\p{Number}] [\\p{Letter}\\p{Mark}\\p{Number}\\p{Join_Control}]*) * (? : \\ . [\\p{Letter}\\p{Number}] [\\p{Letter}\\p{Mark}\\p{Number}\\p{Join_Control}]* (? : -+ [\\p{Letter}\\p{Number}] [\\p{Letter}\\p{Mark}\\p{Number}\\p{Join_Control}]*) *) * $/u`.

This regular expression uses Unicode property escapes and is easier to read as `/^LD (? : -+ LD) * (? : \\ . LD (? : -+ LD) *) * $/u`, where LD is `[\\p{Letter}\\p{Number}] [\\p{Letter}\\p{Mark}\\p{Number}\\p{Join_Control}]*`. If your input matches the regular expression, I lowercase your input, NFKC-normalize it, and make sure that the Unicode normalization has introduced no additional dots, such as 1_ → 1. After Punycode encoding the internationalized domain, I also check that each label of the domain name consists of at most 63 characters. If you have a suggestion for how I can improve my validation, let me know.

▼ Homograph attack

Domain names which look identical but resolve to different addresses are a serious security issue. For example, the lowercase letter l, the uppercase letter I, and the number 1 can easily be mistaken for one another depending on the font, and so can the capital letter O and the number 0. While the problem already existed with ASCII-only domain names, internationalized domain names made the situation considerably worse. For example, the Latin B, the Greek B, and the Cyrillic B all look the same. While BBC.com takes you to the website of the British Broadcasting Corporation (BBC), BBC.com takes you to a completely different website. Deceiving users with optically similar characters in order to obtain sensitive information is known as a homograph attack. While phishing cannot be fully eliminated, such attacks can be mitigated by the client, the registry, and the user:

- **Client:** Browsers and mail clients should warn the user about suspicious domain names and display such domain names in Punycode/ASCII rather than Unicode. Domain names are suspicious when they use characters which don't belong to the user's preferred language or when they mix characters from different scripts. Additionally, it's a good idea to lowercase and normalize domain names before displaying them in a font which clearly distinguishes between visually similar characters.
- **Registry:** Domain name registries should develop registration policies for their top-level domains. Registries are free to permit characters only from certain scripts or not to support internationalized domain names at all. For example, the Russian top-level domain .рф permits only subdomains in the Cyrillic script. Registries which allow the use of different scripts should ensure that the different scripts cannot be mixed in a single label. The Unicode Technical Standard 39 with its data set contains more information about confusable characters. On top of this, registries should bundle or block variants of the same word as outlined in RFC 4290. Wikipedia lists which top-level domains support IDNs and which top-level domains are internationalized themselves.
- **User:** Users should be trained to recognize phishing attempts and to always enter the address of important online services themselves instead of following a link. In the above example, the fact that BBC.com looks just like BBC.com is not a problem if users enter the perceived address into the address field rather than copying it there.

▼ Email address internationalization (EAI)

So far, we have seen how non-ASCII characters can be encoded in the message body, in header fields and in domain names. The only thing that is missing is the internationalization of the local part of email addresses. This is achieved by the following RFCs, which extend the email protocols and the message format to allow Unicode characters encoded in UTF-8 everywhere:

- **RFC 6530** introduces the framework for internationalized email. It explains the problem and defines the used terminology. Unlike earlier proposals, internationalized messages are no longer downgraded in transit because the local part of an address is to be interpreted only by the host specified in the domain part of the address. If an intermediary mail server doesn't support UTF-8, the message has to be returned to the sender. If an internationalized message shall be delivered to legacy mail servers, it has to be downgraded before or during

message submission. Additionally, the incoming mail server of the recipient may downgrade messages after the final delivery so that they can be retrieved by legacy mail clients of the recipient (see the points below). The RFC recommends that incoming mail servers normalize the local part of an email address ideally to NFKC but at least to NFC as part of the address normalization. Senders, however, should not normalize the addresses of recipients.

Email service providers which provide their service to the general public need to be aware that allowing Unicode characters in the local part of email addresses makes it easier to impersonate their users with homograph attacks. Just as domain name registries, public email service providers should either restrict the permitted characters to ASCII or a single Unicode script. Otherwise, they should bundle or block addresses with confusable characters. Other than domain names, which are case-insensitive, email service providers may (but should not) distinguish between different addresses based on the capitalization of the local part. Therefore, mail clients cannot lowercase the local part before displaying it even though this would help to tell characters such as capital i and lowercase L apart.

- RFC 6531 defines an SMTP extension with the keyword SMTPUTF8. If the SMTP server indicates this capability, the SMTP client can transfer a UTF-8 message with UTF-8 envelope addresses by using the MAIL FROM command with the SMTPUTF8 parameter. This RFC also defines additional protocol types, which can be used in the with clause of Received header fields.
- RFC 6532 extends the syntax rules of RFC 5322 to allow the use of UTF-8 characters everywhere. It also introduces an additional content type with the identifier message/global to describe internationalized messages encoded in UTF-8.
- RFC 6533 brings UTF-8 to delivery status notifications (DSN), such as non-delivery reports (NDR).
- RFC 6855 specifies an IMAP extension which allows mail clients to access internationalized messages (and to use Unicode characters in folder names). The UTF8=ACCEPT capability indicates that the IMAP server supports UTF-8 in strings. The UTF8=ONLY capability indicates that the IMAP server requires UTF-8 support from clients because it won't downgrade internationalized messages for them. The UTF8=ONLY capability implies the UTF8=ACCEPT capability and clients have to indicate that they can handle UTF-8 by sending . ENABLE UTF8=ACCEPT to the server.
- RFC 6856 specifies a POP3 extension to upgrade an ASCII-only session to an UTF-8 session. The POP3 server indicates that it supports UTF-8 with the UTF8 capability. A POP3 client can then enable the UTF-8 mode with the UTF8 command. This RFC also introduces a LANG capability and command, which allows the client to configure a different language for the response texts. This can be useful when the client presents error messages from the server directly to the user.
- RFC 6857 specifies an advanced downgrading mechanism for internationalized messages. POP3 and IMAP servers can use it to convert UTF-8 messages to ASCII-only messages before delivering them to mail clients which don't support UTF-8. The conversion is relatively straightforward: Everywhere where the Encoded-Word encoding is allowed, this encoding is used to encode UTF-8 strings as ASCII strings. The Encoded-Word encoding is also used if necessary for unknown header fields. Internationalized domain names are downgraded with the Punycode encoding. Email addresses with non-ASCII characters in the local part are rewritten by encoding the whole address as an Encoded Word and replacing the address with an empty group construct. For example, From: José <josé@example.com> is converted to From: =?UTF-8?Q?Jos=C3=A9_?= =?UTF-8?Q?jos=C3=A9=40example=2Ecom?= : ; thanks to RFC 6854. Since this string encodes an empty group instead of an address, the recipient cannot reply to such a message without manual intervention. RFC 6857 requires the use of UTF-8 as the character encoding and RFC 2047 requires that the @ symbol and the period are also encoded when the Encoded Word precedes an address. If the internationalized email address is part of an address group, the whole group is encoded with this technique because groups cannot be nested. Header fields in which addresses are used but the group syntax is not allowed need to be encapsulated: A header field such as Message-Id is replaced with Downgraded-Message-Id so that its value can be encoded as an Encoded Word. The Received header fields are an exception to this rule: Any clauses with non-ASCII characters are simply removed. Lastly, the message body is left as is, even if the content transfer encoding is 8bit.
- RFC 6858 specifies a simpler downgrading mechanism for internationalized messages, which accepts the loss of information in favor of an easier implementation. Internationalized email addresses are replaced with an invalid address, such as invalid@internationalized.invalid. The original address can optionally be encoded in the display name of the invalid address. The subject field is encoded as an Encoded Word, and all other header fields with non-ASCII characters are simply removed. This RFC also extends IMAP so that the server can indicate to the client which messages were downgraded. In order to prevent permanent loss of information, mail clients shouldn't remove the internationalized message on the server. Automatically removing retrieved messages on the server is especially common among POP3 clients. Another problem is that clients often cache messages indefinitely. Even if the client is upgraded to support internationalized messages, it likely still accesses the downgraded messages from the local message store. Last but not least, downgrading message header fields invalidates DKIM signatures.

Content type

Now that we can encode arbitrary content, we need a way to inform the client how to interpret the decoded content. This is done with the Content-Type header field, which has the following format:

```
Content-Type: {Type}/{[Tree].}[Subtype]{+[Suffix]}[; {Parameter}]*
```

The curly brackets need to be replaced as described below, the content in the square brackets is optional, and the asterisk indicates that there may be several parameters.

The content type is also called media type. IANA maintains a long list of registered media types. A content type consists of:

- **Type:** The primary content type describes the general type of data. If the client doesn't recognize the subtype, it can use this information to decide what to do with the content. If the type is text, for example, it can display the raw data to the user, which wouldn't make sense for binary files. The other top-level media types are image, audio, video, font, model for three-dimensional models, application for application-specific formats, message for email messages, multipart for multipart messages, and example for use in documentation.
- **Tree:** RFC 6838 defines four registration trees in order to keep different kinds of subtypes apart. There is the standards tree, which doesn't use a tree prefix and is reserved for formats specified by a standards organization such as IETF; the vendor tree with the prefix vnd. for proprietary formats; the personal or vanity tree with the prefix prs. for experimental and non-commercial formats; and the unregistered tree with the prefix x. for unregistered and thus only locally used formats.
- **Subtype:** The subtype is the name of the content type. See the list of examples below.
- **Suffix:** A structured syntax suffix can be used to specify the syntax of the media type while leaving the semantics of the data to the subtype. IANA maintains a list of registered suffixes. Examples are +xml, +json, and +zip.
- **Parameter:** Parameters can be used to modify the media type. Each subtype specifies which parameters are required and which ones are optional. Optional parameters assume their default value if they are not provided. If several parameters are provided, their ordering is irrelevant, but each parameter may appear only once. The syntax of parameters is name=value. The best-known parameter is charset to specify the character encoding of text content. IANA maintains the standardized character sets and the values of some parameters.

The type, the subtype, and the parameter names are case-insensitive. RFC 6838 doesn't specify whether the tree and the suffix are also case-insensitive but I assume that this is the case. Whether a parameter value is case sensitive depends on the parameter. The default content type for emails is text/plain; charset=us-ascii. As specified in RFC 1945, HTTP uses the same header field with the same media types.

Example content types: text/csv, text/html, image/png, image/svg+xml, image/vnd.adobe.photoshop, audio/mpeg, video/mp4, font/otf, application/javascript, application/pdf, application/vnd.apple.pages, and application/vnd.ms-excel.

▼ Enriched Text

The ability to send formatted text was first introduced in 1992 with a content type of text/richtext. In order to avoid confusion with Microsoft's Rich Text Format (RTF), the content type was renamed to text/enriched the following year and revised again in RFC 1896. Enriched Text is a markup language with HTML-like tags. Let's look at a simple example:

```
MIME-Version: 1.0
Content-Type: text/enriched; charset=us-ascii

<bold>Roses</bold> <italic>are</italic>
<color><param>red</param>red</color>.
```

An example Enriched Text message. [Click here](#) to use this example in the ESMTP tool above.

This data format has mostly been superseded by HTML and is not widely supported. Apple Mail strips all the tags and displays the text without formatting. Gmail doesn't recognize the format and offers the option to download the content instead. Only Thunderbird displays the text with formatting, but it doesn't support the <color> tag.

▼ HTML emails

Nowadays, most messages are formatted with the Hypertext Markup Language (HTML). The text/html media type is specified in RFC 2854. The message from the previous box looks as follows when it is formatted with HTML:

```
MIME-Version: 1.0
Content-Type: text/html; charset=us-ascii

<html>
  <body>
    <b>Roses</b> <i>are</i>
    <span style="color:red;">red</span>.
  </body>
</html>
```

An example HTML message. [Click here](#) to use this example in the ESMTP tool above.

This example works as intended in Apple Mail, Gmail, and Thunderbird. We'll discuss in the [next box](#) how to style [HTML emails](#). For security reasons, mail clients don't execute sender-provided [JavaScript](#). Gmail and some other email service providers still support [dynamic content](#), though. Furthermore, HTML messages cause serious [privacy issues](#), which I'll cover later.

▼ Email styling

HTML is styled with [Cascading Style Sheets \(CSS\)](#). There are [three ways](#) to add CSS to an HTML page:

```
<html>
  <head>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <span>Hello,</span>
    <span>World!</span>
  </body>
</html>
```

External CSS: Load an external style sheet with a [<link> element](#).

```
<html>
  <head>
    <style type="text/css">
      span {
        color: red;
      }
    </style>
  </head>
  <body>
    <span>Hello,</span>
    <span>World!</span>
  </body>
</html>
```

Internal CSS: Embed the style with a [<style> element](#) inside the [<head> element](#).

```
<html>
  <body>
    <span style="color: red;">Hello,</span>
    <span style="color: red;">World!</span>
  </body>
</html>
```

Inline CSS: Repeat the style with the [style attribute](#) for every element.

The problem with HTML and CSS in emails is that the support for them varies a lot among mail clients. As a sender, you want to make sure that your message is displayed as intended for most of your recipients. This forces you to use only features which are supported by most mail clients. While some mail clients [support external CSS](#), many do not. And while many mail clients [support internal CSS](#) by now, some do not. For this reason, most HTML emails are still sent with inline CSS, which is supported by all mail clients that can display HTML emails. By the way, you don't have to inline the CSS manually, there are [tools for that](#).

Since [Gmail started supporting internal CSS](#) in 2016, all [major mail clients](#) support it. It seems that we can finally stop inlining styles in emails. Unfortunately, the situation is still worse than it seems. [Webmail clients](#), such as [gmail.com](#), [yahoo.com](#), and [outlook.com](#), embed the HTML of messages inside the HTML of their websites without using an [<iframe> element](#). Instead, they remove the [<html>](#), [<head>](#), and [<body>](#) elements from the message and rewrite the internal CSS so that it applies only within the [<div> element](#) of the message. As [others have noted](#), Gmail does a bad job at this. Let's look at an example:

```
<html>
  <head>
    <style type="text/css">
      a { color: red; }
    </style>
```

```
</head>
<body>
  <a href="https://en.wikipedia.org/wiki/Roses_Are_Red">Roses are red</a>.
</body>
</html>
```

Styling the color of links. [Click here](#) to use this example in the [ESMTP tool](#) above.

When you send the above message to your Gmail account and view the message in your browser, you'll see that the link is colored in Gmail's default blue. If you inspect the [<a> element](#) with the [developer tools](#) of your browser, you'll see why:

```
/* Gmail's default style: */
.ii a[href] {
  color: #15c;
}

/* Rewritten custom style: */
.msg443033220466499195 a {
  color: red;
}
```

The styles that Gmail applies to the link in the above message.

The problem is that the [specificity](#) of Gmail's default style is higher than the one of the custom style because of the [attribute selector](#). Using `a[href]` yourself doesn't work because [Gmail supports only class, element, and ID selectors](#). You can solve the problem either by using [!important](#) on your style (`a { color: red !important; }`) or by wrapping the message with an ID:

```
<html>
<head>
  <style type="text/css">
    #body a { color: red; }
  </style>
</head>
<body>
  <div id="body">
    <a href="https://en.wikipedia.org/wiki/Roses_Are_Red">Roses are red</a>.
  </div>
</body>
</html>
```

How to solve Gmail's link rendering problem. [Click here](#) to use this example in the [ESMTP tool](#) above. While the [ID](#) and the [descendant](#) selectors are supported by almost everyone, the [child selector](#) is not.

Since [inline styles are more specific](#), you can keep inlining your CSS styles to avoid this problem. Yahoo.com and Outlook.com do a better job than Gmail. Yahoo rewrites the custom style to `#yiv4554178645 a`, which overrides the default `.msg-body a`. Outlook.com has no default style for the `<a>` element, but interestingly, it inlines a brightened color when you activate its [dark mode](#). Personally, I hope that we can abandon inline CSS for HTML emails soon. Since emails are [compressed](#) neither in transit nor in storage, inline styles increase the size of messages significantly. It also makes HTML messages harder to read in their [raw form](#). Moreover, [media queries](#) are not allowed inside the [style attribute](#), which means that you cannot implement [responsive design](#) and optional [dark mode](#) with inline CSS. As we'll see later, Gmail removes the `<style>` element when [quoting an HTML message](#), though. And since email styling can be abused to make the same message [appear differently](#) to different recipients, it might be a good idea to abandon [HTML emails](#) altogether.

▼ Email markup

Since emails can be written in HTML, you can annotate your message so that not just the receiving user but also the receiving client can make sense of it. Making information easier to understand and to act upon for machines is the goal of the [Semantic Web](#). [Gmail supports](#) certain [schema.org ontologies](#). You can use them to define [actions](#), such as one-click confirmations, and [highlights](#), such as reservations, which users can take or view directly from their inbox without even having to open the email.

In order to use these features, senders have to [register with Google](#) and authenticate their messages with [SPF](#) or [DKIM](#). No registration is required for [sending messages to yourself](#), but such messages still have to fulfill the SPF or DKIM requirement. Unfortunately, Gmail doesn't authenticate messages with SPF or DKIM when you send them to yourself, no matter whether you submit them from the web interface or from the command line with the [tool above](#). I got the email markup to work only by following [this tutorial](#).

▼ Dynamic content

[Gmail](#), [Mail.ru](#), and [Yahoo Mail](#) support interactive messages based on [Accelerated Mobile Pages \(AMP\)](#). AMP was initiated by Google in order to make websites load faster on mobile. It achieves this by providing [built-in components](#) and making websites [cacheable](#) so that they can be served through a [content delivery network \(CDN\)](#). While [AMP for websites](#) supports [custom JavaScript](#), you can use only the [default library](#) when using [AMP for emails](#). It's basically a whitelisted [web framework](#). As with [email markup](#), you have to [register yourself as a sender](#) with the email service providers before they display your dynamic content to their users. AMP messages must be [authenticated](#) and must contain an ordinary [HTML or plaintext version](#) of the same content, which is displayed when the mail client is offline or [30 days after receiving the message](#).

▼ Soft line breaks

The problem with [text-based protocols](#) is that some characters, which can usually also appear in the message, are used for a special purpose. In order not to break the protocol, such characters have to be [escaped](#) inside the message. We have seen several examples of this already: [SMTP](#) and [POP3](#) require leading periods on a line to be [escaped with an additional period](#), [IMAP](#) and [Sieve](#) require double quotes and backslashes to be escaped with a backslash in quoted arguments, the [Quoted-Printable encoding](#) requires = to be escaped as =3D, and the [Encoded-Word encoding](#) requires ? to be escaped as =3F. There are two more situations in which email conventions conflict with user intent: The [line-length limit](#) requires the mail client to insert additional newlines and [message quoting](#) with the greater-than sign requires that unquoted lines don't start with >.

[RFC 3676](#) addresses both issues. Firstly, the sending mail client removes spaces before user-inserted newlines and ensures that there is a space before client-inserted newlines. Secondly, the sending mail client inserts a space at the beginning of all newly written lines which start with a greater-than sign. It then informs the receiving mail client about these transformations by adding the `format=flowed` parameter to the `text/plain` content type. When the receiving mail client sees this parameter, it removes the newlines which are preceded by a space and the spaces which are followed by a greater-than sign at the start of a line after determining which lines are quoted.

The sending mail client can either insert [soft line breaks](#) after existing spaces or insert the preceding space as well. It indicates the former behavior by adding the content type parameter `delsp=no`. If the mail client also inserted the preceding space, it adds `delsp=yes`. In the former case, the receiving mail client replaces `SP+CR+LF` with `SP`. In the latter case, the receiving mail client simply removes all occurrences of `SP+CR+LF`. This is useful for content which doesn't use the ASCII space character.

If we reduce the line-length limit to eight (including the [two newline characters](#)), `2 + 2 > 3` can be encoded as follows:

```
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Type: text/plain; charset=us-ascii; format=flowed; delp=no
2 + 2
> 3
```

How to respect the [line-length limit](#) and [message quoting](#) without having to resort to [Quoted-Printable encoding](#).

In order to make [whitespace](#) visible, I've replaced each [space](#) with `␣` and marked each [newline](#) with `↵`.

Note that the leading space on the last line is required. Otherwise, this line would be a quoted 3.

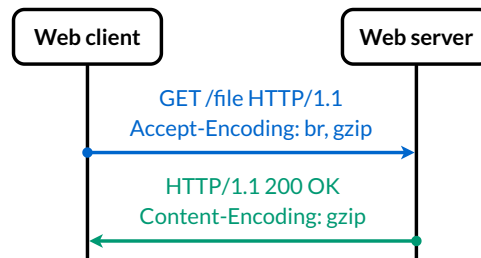
The standard for `format=flowed` is a bit more complicated than how I explained it. On the one hand, a space can be inserted at the beginning of any line, which means that lines which already start with a space have to be protected with an additional space. (For [historical reasons](#), lines which start with `From` also have to be protected by inserting a leading space.) On the other hand, it also specifies how to handle consecutive lines with [different quote levels](#), which always lead to hard line breaks.

As far as I can tell, most messages are encoded with either [Quoted-Printable](#) or [Base64](#), which have their own ways of handling client-inserted newlines. Most users never encounter the line-length limitation of email: They see neither unwanted newlines nor a horizontal scrollbar because a message is displayed as `format=fixed` when it should be flowed. There are exceptions, though. For example, Thunderbird breaks lines in the compose window by default, even though it uses `format=flowed` correctly and displays such messages correctly (i.e. flowed according to the width of your screen). The only fix I found for this annoyance is to set `mailnews.wraplength` to 0 in Thunderbird's [config editor](#). If a line becomes longer than 1'000 characters, Thunderbird then decides to encode the message with Base64 rather than to apply `format=flowed` as it would otherwise.

▼ Message compression

To the best of my knowledge, emails are rarely compressed even though this would save a lot of bandwidth during relay and a lot of memory during storage. For example, HTML newsletters are often between 50 KB and 120 KB in size, and compression reduces this by 70 to 90% due to the many repetitions of styling information. Most of the websites you visit are delivered to your browser in compressed form. So why is it that emails are rarely compressed? The problem in the case of email is that the sender doesn't know whether the mail clients of the recipient support a new encoding. Since compression wasn't included when the now widely supported content encodings were introduced, it's very difficult to transition to compressed message bodies now. Additionally, the email protocols in general and the Quoted-Printable encoding in particular are designed to be human-readable. Compressed data, on the other hand, would have to be encoded with Base64 and can be decompressed only with specialized software. As we will see in the next subsection, it's much easier to introduce new content types than to change the encoding of existing ones because the new and potentially unknown content type can be complemented with known types. Since the goal of compression is to decrease the size of messages, we don't want to add yet another part to messages, though.

In the case of HTTP, we don't have this problem because clients can simply list the encodings that they accept in their requests:



How HTTP compression works. The most used compressions are gzip and br. You can inspect these header fields with the developer tools of your browser.

The only standardized way to compress emails during relay is with S/MIME, which we'll discuss later. Section 3.6 of RFC 8551 provides the following example for how to use S/MIME for compression only:

```
Content-Type: application/pkcs7-mime; smime-type=compressed-data; name=smime.p7z
Content-Transfer-Encoding: base64

eNoLycgsVgCi4vzcVIXixNyCnFSF5Py8ktS8Ej0AlCkKVA==
```

How S/MIME can be used to compress messages using the ZLIB compression format as specified in RFC 1950.

You can decompress this example message with the following commands. pigz stands for Parallel Implementation of GZip, and it can be installed on macOS with `brew install pigz` if you have Homebrew. Using `gzip -dc` instead of `pigz -d` doesn't work because `gzip` doesn't recognize the compression format if the file doesn't start with specific bytes.

```
$ echo "eNoLycgsVgCi4vzcVIXixNyCnFSF5Py8ktS8Ej0AlCkKVA==" | openssl base64 -d | pigz -d
This is some sample content.
```

The pipeline of commands to Base64-decode and ZLIB-decompress the string from the above example message.

Since mail clients have no (standardized) way to advertise their capabilities to other mail clients, we won't see compression of messages from the mail client of the sender to the mail client of the recipient anytime soon. This doesn't prevent mail clients and mail servers from compressing messages between them, though, as they can advertise their capabilities as part of the used protocol. RFC 4978 specifies the COMPRESS extension for IMAP, which allows the client and the server to agree on compressing their communication. Since mail clients can store messages in whatever format they want, compressing locally stored emails is by far the lowest hanging fruit. However, neither Thunderbird nor Apple Mail compresses the messages which it stores locally.

▼ Internationalized parameter values

As we will see in the next subsection, MIME parameters are sometimes used to convey user-chosen information, such as the filename of attachments. RFC 2231 specifies yet another encoding to support non-ASCII characters in parameter values. The sender indicates that the value is encoded by putting an asterisk before the equals sign. The value itself then starts with the name of the character set, followed by optional language information and the encoded value. These three pieces of information are separated by a single quote. The encoding is similar to the percent encoding but the reserved characters which have to be encoded are different and the character set doesn't have to be UTF-8. See the formal syntax in RFC 2231 and RFC 2045 for more information. I've implemented a tool which encodes and decodes extended parameters for you. Let's look at an example:

Decoded: filename="¡Buenos días!.txt"

Encoded: filename*=iso-8859-1'es'%A1Buenos%20d%EDas!.txt



Remarks on the above tool:

- **Invalid inputs:** In both directions, the tool doesn't complain about invalid inputs and simply encodes or decodes the input as good as it can. If your input is invalid, the output of the tool is likely also invalid. Therefore, don't use this tool in production!
- **Quoted values:** Unencoded parameter values have to be quoted if they contain certain characters such as spaces or what the standard calls specials. The quotes themselves are not part of the value, and encoded values are never quoted.
- **Parameter continuations:** [RFC 2231](#) also introduced a mechanism which allows you to split long parameter values in order to adhere to the line-length limit. You can use several parameters to encode a single parameter value by enumerating the parameter name with a decimal number after an asterisk. For example, `name*0="Hello,...":name*1=World!` decodes to `name="Hello, World!"`. Please note that the above tool only decodes parameter continuations but doesn't generate them.
- **Parameter ordering:** According to [RFC 2045](#), the ordering of parameters is not significant. In order to remain backward compatible, this is also the case for parameter continuations: `name*1=World!:name*0="Hello,..."` also decodes to `name="Hello, World!"`.
- **Combining encodings and continuations:** You can combine value encodings and parameter continuations. Encoded and unencoded segments can be mixed. The first segment has to be encoded and it contains the character set and the language information for all the encoded segments. Further encoded segments don't repeat the character set and the language information, which means that you cannot mix character sets with parameter continuations. For example, `name*2*=d%EDas!:name*0*=iso-8859-1'es'%A1:name*1="Buenos..."` decodes to `name="¡Buenos días!"`.
- **Parameter separations:** Not only do you have to reorder continued parameters, you also can't just split the parameters at the semicolons because semicolons are allowed in quoted strings. For example, `name="\";";"` is a single parameter and has to be encoded as such.
- **Language information:** The format of the language tag is specified in [RFC 5646](#). The language tag has been introduced to provide context for screen readers. The language information can be skipped, but the delimiting single quote must be kept. Since the above tool doesn't know in which language you write, it always skips this field when encoding your input. [RFC 2231](#) also extends the Encoded-Word encoding so that other header fields can also specify the language they're written in. An example is `=?US-ASCII*EN?Q?Taste?=?` versus `=?US-ASCII*DE?Q?Taste?=?`, which should be pronounced differently.
- **Unicode normalization:** Everyone normalizes Unicode to NFC except Apple, who thought differently and normalizes filenames in macOS to NFD. If you send a file called `¡Buenos días!.txt` with Apple Mail, the filename is encoded as `filename*=utf-8'!'%C2%A1Buenos%20d%C3%81as%21.txt`. `i%CC%81` encodes the Latin small letter i followed by the combining acute accent. You don't even need to send an email to verify this: You can just paste an NFC normalized string into a filename on macOS and then copy the filename back to the Unicode normalization tool with the normalization option `None` to see that the string is now normalized to NFD.

Multipart messages

Now that we can send arbitrary files via email, we can design file formats to include several files in a single message body. [RFC 2046](#) defines various content types to split a message into multiple parts. What all the multipart formats have in common is that they are text-based. This means that the various parts have to be separated with a character sequence which may not appear in any of the parts themselves. The character sequence is chosen by the sending mail client for each message and provided to the recipient in a content-type parameter called boundary. Let's look at the two most common multipart types and leave the rest for the boxes below.

- multipart/mixed bundles independent parts into a single message. This content type is used to attach files to a message. If a client doesn't recognize a multipart subtype, it should treat the content as `multipart/mixed` and show the recognized parts.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: text/plain
Content-Transfer-Encoding: 7bit

This message has an attachment.

--UniqueBoundary
Content-Type: image/png
Content-Transfer-Encoding: base64

iVBORw0KGgoAAAANSUhEUgAAABQAAAAUCAYAAACNiR0NAAAACXBIXMMAAASAAAL
EgHS3X78AAAB4kLEQVQ4y5VVwU7CQBdKl5IVz8Fg95ZQH+BhBP/UEAxKv/AN3jG
/1ATPeqLJr169WRpi/NmZ8uCEGqTl530zrz07Mx0VdiKFB6s2gwDkeuEAeGRKBaw
```

```
gkR02KvDFj4+hXMcPqUq5T4h1d7LU3Zu1bo+X5GQBGToC2XzC1vML11mtPGMiciQ5
I/xgBRmtBSFHpPS+000rRzzz/JWf5kxf3CKSQneusea8whGyjbIQ4kI+LMHkTou
TpMjA5kZpoQpoUnoEeJ1UikZdiDKmdRG2ldeAWI+HxvZvfIeem9wihKh009EKWuG
D4KDC4WKITpJAUBnQnREugOR3+RjWVkgkMkR8AdtLAMQzj1C4ExIHNkh4XkbIaff
YlJH0Adhos3IpbCN8IDw4hHmshZeoXlpjERJG7jNfYSpd9ZloUij52mixT8IqdId
rvYX4bXsVVLRYRVUn46vryD0wPhRPSnrqVC+Hkp7ytKwFU2w29CKK1Wk70e0seN
8otSpW0+C09MZqIa9kSP5s85QssxqNrYLvrGxt7UXs/xqrGrXb2xmzqx6Jpik6IX
Jbq+8i90heGQs+zvwXZz0FQdX+7ess5EmZ2Nk7/ja/+AHXkDdiQDduL0bPeA3fEL
mMvYTWwJ6Hb+An4Bgrjq/fe5+zgAAAAASUVORK5CYII=
```

--UniqueBoundary--

An example multipart/mixed message. [Click here](#) to use this example in the [ESMTP tool](#) above.

- [multipart/alternative](#) bundles alternative versions of the same content into a single message. This content type is used to provide a fallback version of the content for mail clients that don't support the preferred content type. The versions are to be listed in increasing order of preference, which means that the preferred format comes last. This has the advantage that users of mail clients which don't support multipart messages see the simplest version of the message first. Mail clients usually display the last part which has a content type that they support unless the user configured a different preference. [multipart/alternative](#) is most commonly used to provide a plaintext version of [HTML messages](#) for users of [text-based](#) mail clients, such as [Elm](#), [Pine](#), and [Mutt](#), which [cannot render HTML](#). To give you another example, I could have included a plaintext version of the [Enriched-Text message](#) so that Gmail could display that instead of offering me to download the unrecognized content.

```
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="UniqueBoundary"
```

```
--UniqueBoundary
Content-Type: text/plain
```

Roses are red.

```
--UniqueBoundary
Content-Type: text/enriched
```

```
<bold>Roses</bold> <italic>are</italic>
<color><param>red</param>red</color>.
```

```
--UniqueBoundary
Content-Type: text/html
```

```
<html>
  <body>
    <b>Roses</b> <i>are</i>
    <span style="color:red;">red</span>.
```

```
--UniqueBoundary--
```

An example multipart/alternative message. [Click here](#) to use this example in the [ESMTP tool](#) above.

Since [multipart/mixed](#) and [multipart/alternative](#) are content types like any other, they can be nested, which results in a [tree](#) of message parts. The [content encoding](#) of multipart parts [has to be](#) 7bit, 8bit, or binary, and the [boundary](#) between the inner parts has to be different from the boundary between the outer parts.

▼ Boundary delimiter

For all subtypes of the [multipart content type](#), the sender has to provide a boundary value; there's no default value for this parameter. As mentioned [earlier](#), the double quotes are not part of the value and are required only if the value contains [certain characters](#). The boundary value has to consist of [1 to 70 ASCII characters](#), and it may not end with [whitespace](#). Unlike other parameter values, multipart boundaries [are case-sensitive](#). The various parts are then separated by two hyphens followed by the boundary value and optional whitespace on a line of their own. The [newline characters CR+LF](#) which start and end the [boundary delimiter line](#) belong to the delimiter. As a consequence, the preceding content has no trailing newline characters if there's no empty line between the content and the boundary delimiter line. [More formally](#), the various parts are separated by {CR}{LF}--{BoundaryValue}{OptionalWhitespace}{CR}{LF}. The same line without the leading {CR}{LF} is used to mark the beginning of the first part, and the same line with two hyphens inserted after the BoundaryValue is

used to mark the end of the last part. (If the first line required the leading {CR}{LF}, then the above examples would be wrong because the empty line is used to separate the header from the body and the content itself therefore starts with --UniqueBoundary{CR}{LF}.) Any text before the first part belongs to the preamble and any text after the last part belongs to the epilogue of the multipart content. Both the preamble and the epilogue are ignored by clients which support MIME. Historically, the preamble was used to inform users of clients without MIME support that the rest of the body is a multipart message. Nowadays, the preamble and the epilogue are best used to leave a note to users who know how to inspect the raw message. 😊

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="UniqueBoundary"

Preamble, which is ignored by MIME-supporting clients.

--UniqueBoundary

Part 1 with an implicit content type of text/plain.

--UniqueBoundary
Content-Type: text/plain; charset=us-ascii

Part 2 with an explicit content type of text/plain.

--UniqueBoundary--

Epilogue, which is ignored by MIME-supporting clients.
```

The various parts of a multipart message. [Click here](#) to use this example in the [ESMTP tool](#) above.

Each part starts with zero or more Content-* header fields followed by an empty line and the content of that part. The sending mail client has to make sure that the boundary delimiter line doesn't appear in the embedded content. Due to the leading two hyphens, the delimiter cannot appear in Base64-encoded content. By including =_ in the boundary value, the delimiter also cannot appear in Quoted-Printable-encoded content. The rest of the boundary value is usually chosen randomly. Apple Mail, for example, chooses Apple-Mail=_ followed by a universally unique identifier (UUID) as the boundary value.

▼ Content disposition

If the example from the [previous box](#) is not blocked by your spam filter, your mail client likely displays the content of the second part below the content of the first part. If you want some parts of a message to be displayed as a file, which the user has to open to see its content, you can indicate this with Content-Disposition: attachment. The Content-Disposition header field is specified in [RFC 2183](#). Besides asking mail clients to display a MIME part as an attachment, you can also ask them to display its content inline, i.e. visible between the other parts. With the filename parameter, the sender can suggest a filename for when the recipient wants to store the part in a separate file. If the filename includes non-ASCII characters, it has to be encoded with the Extended-Parameter encoding. The receiving mail client should make sure that the filename conforms to local filesystem conventions and that no file is overwritten without user consent when saving the attachment. The receiving mail client should also ignore any path delimiters in the filename. Let's look at an example:

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: text/plain; charset=us-ascii

Hello, I've attached the source code of the exercise.

--UniqueBoundary
Content-Type: text/plain; charset=us-ascii
Content-Disposition: attachment; filename=HelloWorld.c

#include <stdio.h>

int main() {
    printf("Hello, World!");
    return 0;
}

--UniqueBoundary--
```

For [historical reasons](#), many mail clients also include the filename with the name parameter in the Content-Type header field. If the filename contains non-ASCII characters, they use the [Encoded-Word encoding](#) there contrary to what [RFC 2231](#) specifies.

[RFC 2183](#) specifies additional parameters, which are less commonly used: [creation-date](#), [modification-date](#), [read-date](#), and [size](#), where the size is provided in bytes and the dates are formatted according to [RFC 5322](#) just like the Date header field.

▼ Aggregate documents

Certain content types, such as [HTML](#), can reference external resources, such as styles and images. If the external resources are not attached to the message, your mail client has to fetch them over the Internet in order to display the message as intended by the sender. However, [remote content](#) compromises your privacy and violates the principle that the sender of a message can no longer affect its content once it has arrived in your inbox. These issues can be avoided by attaching all the referenced resources to the message and then referencing these attachments from the main part. The following three standards allow us to do this:

- [RFC 2387](#) specifies the multipart/related [content type](#), which indicates that the various parts are related to one another and should be rendered as a whole. Unless declared otherwise with the [start parameter](#), the main part, which references the other documents, is the first part of the multipart/related content. The [type parameter](#) has to be set to the content type of the main part. The [content disposition](#) of the other parts is [relevant only](#) for mail clients which don't recognize the multipart/related content type and render the parts as multipart/mixed.
- [RFC 2045](#) specifies the Content-ID header field to identify the parts of a multipart message. The syntax of the [Message-ID header field](#) is used for the Content-ID header field, and just like the Message-ID, each Content-ID has to be generated in such a way that it is globally unique.
- [RFC 2557](#) specifies two ways how other MIME parts can be referenced from HTML. The first way is to use cid as the [scheme name](#) of [Uniform Resource Locators \(URLs\)](#) to refer to the Content-ID of another part. The disadvantage of this approach is that the URLs of an existing HTML page have to be rewritten. For this reason, the second way leaves the HTML page as is and introduces a [Content-Location header field](#) instead. This header field can be used in a referenced part with the URL that is used to reference it in the main part. The URL should be globally unique, but it doesn't have to resolve to a document, and it may even resolve to a completely different document. What makes the second approach a bit more complicated is the potentially required [encoding of the URL](#) and how [relative URLs are resolved](#). The resulting multipart/related format, which is called [MHTML](#), is also used to [archive websites](#) independently from email.

You can find [plenty of examples](#) with [plenty of errors](#) in [RFC 2557](#). Here is an example of my own:

```
MIME-Version: 1.0
Content-Type: multipart/related; boundary="UniqueBoundary"; type="multipart/alternative"

--UniqueBoundary
Content-Type: multipart/alternative; boundary="InnerBoundary"

--InnerBoundary
Content-Type: text/plain; charset=us-ascii

https://ef1p.com

--InnerBoundary
Content-Type: text/html; charset=us-ascii

<html>
  <body>
    <a href="https://ef1p.com" style="text-decoration: none; font-weight: bold; color: #0D4073;">
       ef1p.com
    </a>
  </body>
</html>

--InnerBoundary--

--UniqueBoundary
Content-Type: image/png
Content-ID: <logo@ef1p.com>
Content-Transfer-Encoding: base64
Content-Disposition: inline; filename=logo.png

iVBORw0KGgoAAAANSUheUgAAABQAAAAUCAyAAACNiR0NAAACXBIWXMAAAASAAAL
```

```
EgHS3X78AAAB4kLEQVQ4y5VVwU7CQBDdK15IVz8Fg95ZQH+BhBP/UEAxKv/AN3jG
/1ATPeqLJr169WRpi/NmZ8uCEGqTL530zrz07Mx0VdiKFB6s2gwDkeuEAeGRKBaw
gkR02KvDFj4+hxMCpUq5T4h1d7LU3Zulbo+X5GQBGT0C2XzC1vML1lmtPGMiciQ5
I/xgBRmtBSFHpPS+000rRzzz/JWf5kxf3CKSQneusea8whGyjbIQ4ki+LMHHkTou
TpMjA5kZpoQpoUnoEeJ1UiKzdiDKmdRG2ldeAWI+HxvZvfIeem9wihKh009EKWuG
D4KDC4WkITpJAUBnQnREugOR3+RjWVkgkMkR8AdtLAMQzj1C4ExIHNkh4XkbIaff
YlJH0Adhos3IpbCN8IDw4hHmshZeoXLPjERJG7jNfYSpd9ZloUIj52mixT8IqdId
rvYX4bXsxVVLlYRVUn46vryD0wPhRPSnrqVC+Hkp7ytKwFU2w29CKK1Wk70e0seN
8otSpW0+C09MZqIa9kSP5s85QssxqNrYLVrGxt7UXs/xqrGrXb2xmzqx6Jpik6IX
Jbq+8i90heGQs+zvwXZz0FQdX+7ess5EmZ2Nk7/ja/+AHXkDdiQDduLobPeA3fEL
mMvYTWJ6Hb+An4Bgrjq/fe5+zgAAAAASUVORK5CYII=
```

--UniqueBoundary--

`` references the part with Content-ID: `<logo@ef1p.com>`.

[Click here](#) to use this example in the [ESMTP tool](#) above.

The parts can also be aggregated differently. Click on the list title to use the corresponding variant in the [ESMTP tool](#) above.

- **Relative Content-Location:** Use `` in the HTML part and Content-Location: `logo.png` in the image part. Apple Mail, Outlook.com, and Yahoo Mail fail to display the message correctly. Only Gmail and Thunderbird implement this part of the RFC.
- **Absolute Content-Location:** Use `` in the HTML part and Content-Location: `https://ef1p.com/logo.png` in the image part. Only the same mail clients as before display the message correctly.
- **Nesting related in alternative:** Instead of nesting the parts as `related(alternative(Plain, HTML), Image)`, you can also nest the parts as `alternative(Plain, related(HTML, Image))`. The mail clients that I tested display the two variants identically. For example, when telling Thunderbird to display the plaintext part, it offered to save the attached image in both cases. As far as I can tell, nesting `multipart/alternative` in `multipart/related` is more common. This structure is also mentioned in [RFC 2557](#), and the second variant doesn't even make it through [Gandi's spam filters](#).

▼ Other multipart subtypes

There are other multipart types for emails besides `multipart/mixed`, `multipart/alternative`, and `multipart/related`:

- `multipart/digest` ([RFC 2046](#)): Include several messages in a single message, which is useful for digests of [mailing lists](#).
- `multipart/report` ([RFC 6522](#)): Complement human-readable ([non](#))-[delivery reports](#) with a machine-processable part.
- `multipart/signed` ([RFC 1847](#)): [Append the signature](#), which is generated over the first MIME part, in the second part.
- `multipart/encrypted` ([RFC 1847](#)): [Prepend the information](#) needed to decrypt the second MIME part in the first part.

After many encoding-related sections, I want to mention two more format-related aspects before moving on to [issues with email](#).

One-click unsubscribe

If you are subscribed to a [mailing list](#), you may want to unsubscribe from the list after having received a message you no longer want to receive. Most mailing lists include a link at the bottom of each sent message, which you can click to unsubscribe from the mailing list. Since this is a link like any other in the message, a browser window is opened and you might have to click on additional buttons there to finally unsubscribe from the list. This can be a bit of a hassle, especially on mobile phones. Fortunately, [RFC 2369](#) specifies an easier way to achieve the same. Mailing lists should include a [List-Unsubscribe header field](#) so that mail clients can provide a uniform unsubscribe experience across mailing lists: You simply click on "Unsubscribe" and your mail client takes care of the rest.

```
List-Unsubscribe: <https://example.com/unsubscribe?token=XYZ>,
<mailto:unsubscribe@example.com?subject=Unsubscribe>
```

The List-Unsubscribe header field provides a standardized way to unsubscribe from a mailing list.
If there are several options in angle brackets, the mail client should use the first one that it supports.

To be precise, [RFC 2369](#) didn't require that there is no additional user interaction. In fact, user confirmation was often necessary in order to prevent accidental unsubscriptions triggered by anti-spam programs which simply fetch all the links in a message. For this reason, [RFC 8058](#) defines an additional header field with more precise semantics: When the user clicks on "Unsubscribe", the mail client sends a [POST request](#) to the HTTPS resource specified in the List-Unsubscribe header field with the value of the new List-Unsubscribe-Post header field in the body of the request. The List-Unsubscribe-Post header field has to contain `List-Unsubscribe=One-Click`, and both header fields [must be covered](#) by a valid [DKIM signature](#).

```
List-Unsubscribe-Post: List-Unsubscribe=One-Click
```

The List-Unsubscribe-Post header field informs the receiving mail client that it can unsubscribe the user with a simple POST request.

The body of the POST request is encoded with the `content type` `multipart/form-data` as specified in [RFC 7578](#) or `application/x-www-form-urlencoded` as specified by the [Web Hypertext Application Technology Working Group \(WHATWG\)](#) in their [URL spec](#). The request has to be sent without context information such as [cookies](#). The user has to be authenticated with a [token](#) in the URL.

```
POST /unsubscribe?token=XYZ HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

List-Unsubscribe=One-Click
```

The POST request to unsubscribe a user from a mailing list [as generated by Gmail](#).
You find an example POST request using `multipart/form-data` in [RFC 8058](#).

These two header fields are not only convenient for users, they also make unsubscribing more secure since mail clients don't include them when forwarding a message. If you want to prevent others from unsubscribing you from a mailing list, you have to remove the unsubscribe link at the bottom of a message manually before forwarding the message.

Custom header fields

[IANA](#) maintains a long list of [registered message header fields](#). The ones specified in an RFC and thus endorsed by IETF are called permanent header fields. The ones registered for [private use](#) without official recognition are called provisional header fields. [RFC 3864](#) outlines the registration procedure for header fields. It's common to start the name of custom header fields with X-, but unlike in the case of `content types`, there is no requirement for this. [RFC 822](#) just promised that official header fields will never start with X-. This provision regarding extension header fields was dropped in [later revisions](#), though. During my research for this article, I've inspected a ton of messages in their [raw format](#). The funniest header field I came across is the following one from [Booking.com](#):

```
X-Recruiting: Like mail headers? Come write ours: https://careers.booking.com
```

This is one way to reach nerds like me. I'm not *that* excited about email headers, though. 😊

Issues

Email is both a blessing and a curse. On the one hand, email is by far the most important decentralized messaging service that we have, which should be reason enough to cherish it. The only other decentralized messaging service which comes close to email in terms of ubiquity is the [Short Message Service \(SMS\)](#). On the other hand, email has become so dysfunctional that many of us would like to leave it behind. In this section, we'll look at the issues that plague modern email. In the [last chapter](#), we'll discuss how some of the security-related issues are being addressed.

Spam

Unsolicited messages which are sent in large quantities are called [spam](#) or [junk mail](#). [Spam](#) is a brand of canned pork, which was introduced in 1937. Spam is likely an abbreviation for spiced ham. It became ubiquitous during and after [World War II](#) when food was rationed. The British comedy group [Monty Python](#) made fun of this fact in a [famous sketch](#) in 1970. The term got adopted to refer to undesirable things which come in excessive quantities – including junk mail.

Any messaging service which is popular, open, and free will have spam sooner or later. Thus, spam isn't a result of the shortcomings of email but rather a consequence of its desirable properties. Since unsolicited messages are annoying, people try to eliminate junk mail from their inboxes with [heuristics](#), [blacklists](#), and [challenges](#). While [such techniques](#) make spam bearable, they don't solve the underlying problem of unsolicited mail: Anyone in the world can add tasks to the to-do list which is your inbox. In my opinion, mail clients should separate messages from unapproved senders from your inbox so that the messages you actually want to receive don't drown in the noise. This is similar to how I almost never accept calls from numbers that I haven't stored in my phone. Even though this feature has to be tremendously useful for anyone who doesn't want to be bothered by random sales people and their never-ending followups, [HEY](#) is the only mail client I know of which let's you screen your email senders. And just like I block call centers, I also block email senders, of course. However, the default shouldn't be "allowed unless blocked" but rather "blocked unless allowed". Additionally, messages are typically blocked on the client-side because most mail clients still don't support [server-side filtering](#).

▼ Heuristics

Labelling messages as spam based on their content and origin is a classification problem. Probabilistic classifiers calculate how likely an incoming message is spam based on statistical models. A popular method is Naive Bayes spam filtering. Since messages are either kept in the inbox or discarded as spam, the continuous probability has to be converted into a binary decision. If the probability that a message is spam is above a certain threshold value, it is discarded. While users hate spam, they hate losing legitimate messages due to spam filters even more. In other words, the rate of false positives has to be close to zero, while the rate of false negatives can be higher. For this reason, the threshold for discarding a message is usually quite high. Messages with a score above a lower threshold are typically moved to a spam folder so that the user can decide what to do with them.

	Is spam	Is not spam
Labelled as spam	True positive	False positive
Labelled as non-spam	False negative	True negative

How binary classifiers are evaluated. When labelling spam, you want to have as few false positives as possible, even if this increases the rate of false negatives as well.

Filtering messages based on heuristics is a cat-and-mouse game: The better the filters, the higher the selective pressure on spammers. While dropping messages without a bounce message violates the principle that messages are either delivered or returned, you don't want to reject messages based on their content as this would offer spammers a valuable test environment.

▼ Blacklists

As long as spammers send their unsolicited messages from the same sources, you can get rid of all their junk simply by blocking all traffic from these sources. Historically, lists of blocked addresses are known as blacklists and lists of allowed addresses as whitelists. Since some consider the positive and negative connotations of white and black to be racially charged, the IT industry is moving to replace these terms with block or deny list and allow or pass list, even though the traditional terms likely predate attribution to race. In the spirit of making the IT industry and our societies more inclusive, I welcome these changes. The main reason why I stuck with the old terms is the next box: The anti-spam technique is known only as graylisting. If I were to speak of temporarily-reject listing, many would have no idea what I'm talking about.

While block lists are already useful when every provider maintains their own list, they are much more powerful when they are shared among email service providers. The best-known maintainer of block lists is The Spamhaus Project. Before you try to relay email directly with the ESMTP tool, you can check here whether your IP address is blocked. If you use and misuse the ESMTP tool a lot, your address may get listed there. Once your address is on their block list, your chances of relaying emails successfully dwindle. Block lists are fed by spam filters, which in turn are trained by users, who mark unwanted messages as spam. Another way to identify spammers is to set up a honeypot: An email server or email address which is positioned to attract spammers but unlikely to be contacted by legitimate parties. Anyone who takes the bait can be blocked.

▼ Graylisting

Similar to requiring a reverse DNS entry, graylisting is a hand-crafted heuristic: Legitimate emails likely pass the hurdle, while spammers fail often enough for the hurdle to be useful. Incoming mail servers can reject incoming mail with a temporary error. (Since you might get graylisted yourself when you use the ESMTP tool: The reply codes for temporary errors start with a 4.) Outgoing mail servers of legitimate parties keep the emails which could not yet be delivered in their queue. Spammers, on the other hand, typically continue with the next address in their list without coming back to the addresses which failed on the first attempt. By rejecting messages from unknown senders temporarily, mail servers can reduce the volume of spam significantly. The disadvantage of graylisting is that it also delays the delivery of legitimate messages. RFC 5321 recommends waiting at least 30 minutes before the next attempt. Emails which aren't delivered instantly are as annoying as spam: If you sign up on a new website or reset your password, you want to be able to continue immediately. Nonetheless, graylisting is the lesser evil.

▼ Patience

There are other areas where the strict enforcement of RFC standards pose a hurdle for impatient spammers:

- **Greeting delay:** SMTP clients should wait for the greeting from the server before sending the EHLO command. Spammers who want to maximize the use of their bandwidth either drop the connection when the greeting is delayed or send the EHLO command immediately, which can be rejected by the server. By delaying the greeting only for unknown senders, you can slow down spammers without affecting everyone. Slowing down fraudulent software is also known as tarpitting.

- **Quit detection:** SMTP clients must send the QUIT command before closing the connection. Since the email is already queued for delivery at this point, some spammers skip this step so that they can open the next connection sooner. When a mail server detects this behavior, it can include this information in its spam assessment.
- **Invalid pipelining:** In order to reduce the number of round trips, many mail servers allow clients to batch their commands. The standard requires, though, that clients wait for the response code of certain commands. As you can see in the ESMTP tool above when you activate pipelining, the client has to wait for the response to the EHL0 command to determine whether the server even supports pipelining and for the response to the DATA command before sending the actual message. Since spammers don't care about errors, they are tempted to send all commands at once. Mail servers can detect invalid pipelining and reject all messages from such senders.
- **Invalid MX records:** SMTP clients should connect to the mail server with the highest priority first. If the server cannot be reached, they should attempt to connect to the mail server with the second-highest priority. Since some spammers don't retry on failure but legitimate senders do, you can point the MX record with the highest priority to a non-existent server. This technique is called notlisting.

▼ Challenges

Spamming is an economic activity and economic activities are worthwhile only if the benefits are higher than the costs. The reason why there is so much spam is because the marginal cost of sending an email is almost zero. If we increase the cost of sending emails by just a little bit, most spammers would go out of business. There are two ways to increase the cost of sending emails in large quantities: You can require manual intervention from unknown senders or force them to waste a costly resource such as electricity, bandwidth, or memory. In both cases, your incoming mail server or your mail client would send a puzzle to the sender, who needs to solve it in order for their email to be delivered to your inbox. To require attention from a human, you can send them a CAPTCHA. To require attention from a machine, you can ask it for proof of its work. Since your challenge is an automatic response, care needs to be taken to avoid mail loops. The main disadvantage is that you confirm the existence of your email address to anyone (unless your mail server sends a challenge for existent and non-existent recipients). Given that you now have an effective system against spam and that many email addresses are public anyway, this shouldn't be a problem in practice. To be honest, I'm surprised that this old and simple idea isn't being used more often. The difficulty of the puzzle could depend on the likelihood of the message being spam. Companies such as online platforms would have to ask their users to whitelist their domain, let them recover the first message from the spam folder – or employ people to solve the puzzles.

▼ Reputation

The origin of a message plays a crucial role when assessing its trustworthiness. In the absence of domain authentication, it's mostly the reputation of the sender's IP address, which determines whether a message gets delivered. Once emails can no longer be spoofed, the reputation of the sender's domain will also become important. When you deliver emails from a new IP address, they will likely land in the spam folder of their recipients even if you follow all best practices. Your IP address just isn't known yet to deliver emails that users want to receive. A good reputation takes time to build, which makes it quite difficult to run your outgoing mail server yourself. Especially as a company, you want all your customers to receive all your emails. In order to achieve a high delivery rate (also called good deliverability), you typically buy into the reputation capital of another company. A whole industry evolved around just this value proposition. Such companies are known as SMTP service providers or email delivery vendors, and they offer a transactional email service. The downside of this reputation system is that email is no longer really an open service if you have to purchase the qualification to send messages from another company. The upside of this system is that companies are incentivized to protect their reputation: They rather want to make it as easy as possible for readers to unsubscribe from their newsletter than to risk being flagged as spam.

▼ Address munging

One of the best ways to avoid getting spammed is to keep your email address as private as possible. Unfortunately, a single hack of one of your service providers is enough to expose your address permanently. Spammers harvest email addresses in various ways. Besides purchasing lists of email addresses on the black market from hackers and other spammers, they use programs, so-called crawlers, which search the Web for email addresses. In an attempt to prevent their address from being collected, people often disguise their address when publishing it online. For example, instead of writing user@example.com, they write user[at]example.com and the like. This practice is called address munging and it is most effective if the particular technique is rarely used or difficult to revert. Unless the address obfuscation is reverted in the browser with JavaScript, readers who want to contact the person cannot simply click on a mailto link. Other approaches such as encoding email addresses as images instead of text reduces the accessibility for people who rely on screen readers.

▼ Legal requirements

Many countries adopted anti-spam laws, which ban unsolicited bulk mailing. While the legal requirements for email marketing vary by country, you're typically restricted to contact only people who gave their explicit consent to being contacted by you. Additionally, you usually have to provide a way to opt out from your mailing list, include the name of your company and its physical mailing address, ensure that the

From and Reply-To address are valid and active, and provide non-misleading content in the subject and body of your email. On top of that, you also have to adhere to privacy acts, such as the European [General Data Protection Regulation \(GDPR\)](#). Since another person can generally sign you up to a [newsletter](#), it's a good practice to ask new subscribers via email to confirm their subscription before sending them the newsletter. This practice is known as [double opt-in](#), and without confirming the email address, you will likely struggle to prove that the recipient gave their explicit consent.




Privacy

If you send an email to someone, you want to share certain information with that person. Mail clients and mail servers, however, share a lot more information than what the users intended to share. In this subsection, I list all the subtle information disclosures that users likely aren't aware of. If you know of other privacy leaks, please [let me know](#).

Sender towards recipients

The recipient of a message often learns the following information about the sender:

- **IP address:** When you submit an email to an outgoing mail server, most mail servers add your [IP address](#) to the message as part of the [trace information](#). As a recipient, you find the IP address from which a message was sent in an x-originating-ip header field or in the square brackets in the first parentheses of the last Received header field. (Each mail server through which an email passes adds an additional Received header field at the top, which means that the first Received header field, which was added by the outgoing mail server, is at the bottom.) There are three important implications of this. Firstly, your outgoing mail server leaks your rough physical location to all recipients. In other words, never send to your boss that you're sick at home from your holiday apartment. Similarly, recipients can tell whether you're still at work or went home already. Secondly, recipients can launch a [denial-of-service attack](#). Due to [network address translation \(NAT\)](#), the target would typically be your router rather than your machine, but your Internet connection goes down either way. Thirdly, if you visited the website of an email recipient anonymously or pseudonymously, the recipient now knows who this user on their website is. To find out whether your outgoing mail server includes your IP address in the messages that you send, send a message to yourself and search for your IP address. You can use the following tool with an empty input field to determine your IP address. You can also use the tool to [locate the IP address](#) of someone who sent you an email. The tool uses the geolocation API of [ipinfo.io](#).

IPv4 address:   

If you don't want your email service provider to leak your own IP address, you can use a [Virtual Private Network \(VPN\)](#) or an [overlay network](#) for anonymous communication, such as [Tor](#). Alternatively, you can use an email service provider which values your privacy, such as [ProtonMail](#) or [Tutanota](#). Sending messages from the [web interface](#) of an email service provider usually also helps. For example, if you compose an email on [gmail.com](#), your IP address is not included in the outgoing message. If you submit a message from your desktop client to Gmail using [SMTP](#), on the other hand, your IP address is added by `smtp.gmail.com` in a Received header field. While [RFC 5321](#) does say that the IP address of the source should be included in the Received header field, email service providers should ignore the standard in this regard, in my opinion. I understand that email service providers may want to record the IP address of the sender to prevent abuse of their service, I just see no reason to share this information with the recipients of a message. In fact, it might even be illegal to do so. Many privacy acts, such as the European [General Data Protection Regulation \(GDPR\)](#), forbid service providers to share personal data without the user's explicit consent. Since the third party with whom the personal data is being shared can be different for every email, the user's consent would be required every time they send an email. If you're a lawyer and you think that this reasoning has some merit, let me know so that we can file a [class-action lawsuit](#) to bring this industry practice to an end.

- **Device name:** Mail servers also include the client's argument to the EHLO command in the Received header field. [RFC 5321](#) requires that the client uses its [fully qualified domain name](#) if it has one or its IP address otherwise. In spite of this, Thunderbird and maybe other clients use the name of your device in the local network as the argument. On macOS, you find the [name of your device](#) in the "Sharing" tab of your "System Preferences". By default, it starts with the first name of your user account. In my case, my computer is reachable under `Kaspar's-MacBook-Pro.local` in the local network. As a [whistleblower](#), I might create a new email address and even use an anonymization service, such as [Tor](#), just to have my mail client and mail server leak my real name. [RFC 5321](#) even warns about exactly this problem. I reported [this privacy bug](#) to Mozilla Thunderbird on 2 December 2020. Until a fix is available, you can set the `mail.smtpserver.default.hello_argument` option in the [config editor](#) to `[192.168.1.1]`. Such a value is typical for the vast majority of people due to [network address translation \(NAT\)](#).
- **Timezone:** The [sent_date](#) is usually encoded in the timezone of the sender. By looking at the offset from the [Greenwich Mean Time \(GMT\)](#), the recipient learns from which [longitude](#) a message was sent. In my opinion, mail clients should always encode the Date field in Greenwich Mean Time.
- **Mail client:** Many mail clients put their name with their current version into a User-Agent or X-Mailer header field. Some mail clients even include the name and the version of the operating system on which they run. While such data is usually harmless, it can provide valuable information to someone who wants to attack you. Given the intricacies of email, mail clients can also be identified by how they [delimit parts](#), how they [label files](#), how they [style messages](#), how they [quote messages](#), and so on. This is known as [fingerprinting](#), and it allows a recipient to determine whether separate messages were sent from the same client.
- **Display names:** Your mail client not just adds your name as a [display name](#) in the From address, it also adds a display name for each recipient it knows. This can leak how you've stored a recipient in your address book (i.e. be careful under what name you store the colleague you're having

an affair with) and with whom of the recipients you've been in contact before (because mail clients usually add display names from earlier conversations automatically). As a recipient, you have to inspect the raw message to see what the sender provided because your mail client typically overwrites the display names with the information from its own address book. In my opinion, mail clients should remove the display names of recipients before sending a message.

- **Hidden recipients:** The Received header field has an optional for clause, which contains the address of the specified recipient. As recommended by RFC 5321, the for clause is skipped when there are several recipients in the envelope of the message. As a consequence, a single non-hidden recipient learns that the message was also sent to hidden recipients if the for clause is missing in the bottommost Received header field. This means that the empty Bcc field approach is used more often than intended.
- **Attachments:** The content disposition of attachments can include information such as when the file was created and when it was last modified. While it can be useful to preserve such information when mailing a file, sharing such information with the recipient can also be unexpected and undesirable. I don't know how mail clients can determine the preferred option without cluttering the user experience. By default, they should err on the side of caution, which many do.

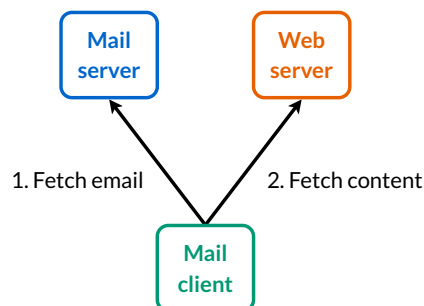
Assuming that all recipients can be trusted is foolish. If someone pretends to be interested in your work, you'll likely reply to them.

Recipient towards sender

There are two ways in which the sender can track the recipient: By including remote content in the message and by redirecting external links. If the sender can trick you into replying to them or your mail client sends a read receipt, then all the above privacy issues also apply, of course.

Remote content

HTML emails can include remote content, which is fetched by the mail client when it renders the message. Images are by far the most common type of remote content. They are usually included with the element or with the background-image property. Some mail clients support external style sheets through the <link> element, but internal CSS can also have @import statements to load Web fonts and other styles with the url() function. There are other elements, such as <audio>, <video>, and <iframe>, which can also be used to include remote content, but not all mail clients support them.



After fetching the email from the trusted mail server, the mail client fetches the remote content from the untrusted web server.

Remote content violates three fundamental principles of email:

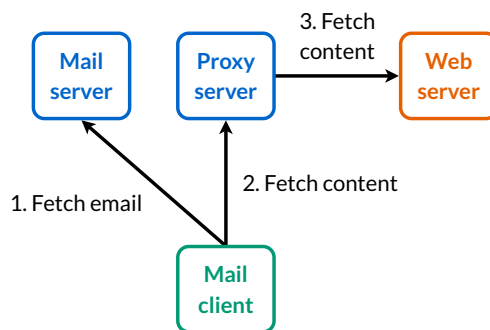
1. **Offline reading:** Since mail clients usually fetch the remote content only when you open the message, substantial parts of the message can be missing when your computer is not connected to the Internet. Since most mail clients don't cache the remote resources, being online when you open the message for the first time isn't enough.
2. **Immutable content:** Most users probably think that once they have received an email, the sender can no longer modify it because your inbox contains an independent copy of the message, to which the sender has no access. Unfortunately, this assumption doesn't hold for HTML emails with remote content. Since remote content isn't cached, different content can be provided every time you open a message. Some clever engineers used this circumstance to include a dynamic Twitter feed in an email. If you're not aware of this "feature", though, you might fall for a scammer who seemingly predicted the development of some market accurately. And even if the remote content isn't modified, you can no longer view the original message once the sender stops hosting it. The situation gets even worse if the domain on which the remote content is hosted is transferred to a new owner or if the web server which hosts the remote content is compromised by an attacker. Furthermore, remote content isn't covered by message signatures. In theory, some of these security issues could be addressed with a technique known as subresource integrity (SRI). If the sender included the hash of the resource in the original message, then the resource could no longer be modified afterwards. Unfortunately, subresource integrity is specified only for <script> and <link> elements. While a future revision of the specification might add support for integrity checks to other elements, there are no plans for this yet.
3. **Reading privacy:** Whenever your mail client fetches a remote resource, the web server operator learns when and from where the resource has been accessed. Since most mail clients include a User-Agent header field in their HTTP request, the web server operator also learns which mail client you use. For the reasons mentioned in the previous point, senders should reference only remote content which they control. Email newsletters often include remote content with a personalized URL just to track who opened the message when and from where. Based on this data, the sender can determine what percentage of recipients opened the email, which is known as the open rate. It's important to note that your privacy when reading emails is not worse than when browsing the Web. The crucial difference is that on the Web, you go to a website,

whereas in the case of email, the website comes to you. Since you don't want to provide your IP address to anyone, you should disable remote content in your mail client.

In my opinion, remote content should never have been supported by mail clients. If people insist on incorporating related files into a message, they can use aggregate documents. Now that remote content is used so widely, we have to live with the above drawbacks.

▼ Proxying remote content

Google and Yahoo proxy all remote content in their webmail clients. Instead of letting your browser fetch the remote content directly, these companies fetch the remote content on your behalf. The advantage of this approach is that your IP address no longer leaks to the sender of a message. Unfortunately, Google and Yahoo fetch the remote content only when you open the email. Thus, you still let the sender know that you've opened the email and when you've opened it. Google seems to cache the external resources for some time. Yahoo, on the other hand, makes another request if you force your browser to reload all content. In order to fully protect the privacy of their users, these companies would have to fetch and cache all remote content as soon as they receive the email. In order not to confirm to the sender which addresses exist, they would have to do this for all incoming messages, even the ones with inexistent recipients and the ones which are discarded as spam. Beyond fetching static content, Google also proxies the requests triggered by dynamic content. Since webmail providers have access to your emails anyway, there is no privacy drawback when these companies fetch the remote content for you. Desktop clients, on the other hand, can fetch emails from any email service provider. If a desktop client were to use a proxy server which is operated by the publisher of the client, the publisher would learn from which server you fetch the remote content even if the communication is encrypted end-to-end like in a virtual private network (VPN). The best solution would be to fetch all remote content through the Tor anonymity network. Unfortunately, I don't know of any mail client which does this. 😞



How remote content can be fetched via a proxy server in order to protect the user's privacy to some degree.

▼ How to disable remote content

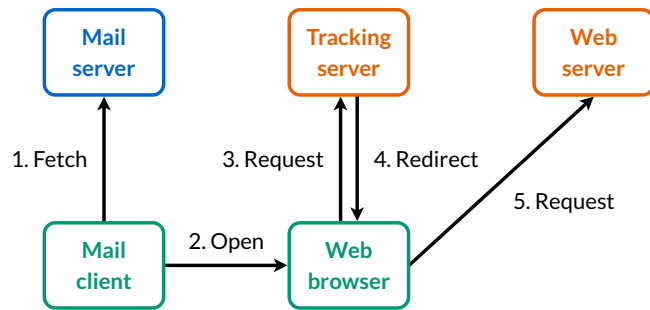
If you care about your privacy, you should allow remote content only from trusted senders. Here is how you disable remote content in various mail clients (where some of them have remote content disabled by default):

- **Gmail:** All settings > General > Images > Ask before displaying external images
- **Yahoo:** More settings > Viewing email > Show images in messages > Ask before showing external images
- **Thunderbird:** Preferences > Privacy & Security > Mail Content > Allow remote content in messages [disabled by default]
- **Apple Mail:**
 - **Mobile:** Settings > Mail > Messages > Load Remote Images [toggle the switch]
 - **Desktop:** Preferences > Viewing > Load remote content in messages [remove the tick]
- **Outlook:**
 - **Web:** View all Outlook settings > General > External images > Always use the Outlook service to load images (As far as I can tell, you cannot disable remote content and the proxy service doesn't work for me.)
 - **Desktop:** File > Options > Trust Center > Trust Center Settings or Automatic Download > Don't download pictures automatically in HTML e-mail messages or RSS items [enabled by default]

Afterwards, you can test your mail client with emailprivacytester.com. After verifying your address, you get an email with 40 different types of remote content, and you can observe in real time which ones are fetched by your client.

Link tracking

Emails often contain links to websites. Instead of linking to the target site directly, the sender can rewrite the link in such a way that your web browser sends a request to their tracking server, which in turn redirects your browser to the actual web server:



How clicks on links can be tracked by the sender of a message.
In general, you cannot trust the servers in orange.

Unlike [tracking pixels](#), link tracking also works in plaintext emails and when remote content is disabled. If the target website isn't identifiable in the tracking link, you have no other choice than to request its address from the tracking server if you really want to see the advertised content. The sender can use tracking links to measure what percentage of recipients opened the link, which is known as the [click-through rate \(CTR\)](#). The same technique is often used on [social media](#) in combination with [URL shortening](#) to determine the reach of a post.

Since seeing is believing, I wrote a little [tool to track emails](#). You can generate a unique token and then subscribe to the associated events below. You can send the tracking link and the tracking image to someone using your mail client or the [ESMTP tool](#) above. I've deployed the tracking server on [heroku.com](#). As you can see in [its source code](#), my server doesn't store anything but [Heroku logs the last 1'500 requests](#), which includes the token, the link, and your IP address. I don't persist the [log file](#), but I might check it from time to time for troubleshooting. In order to determine where a request was made from, the tool uses the free API from [ipinfo.io](#). You can also use the tool to see from where social media apps request an URL to generate a link preview or to convince yourself that the [Tor browser](#) indeed connects from a different location every time you restart it.

Token: Link:

Security

Security and the lack thereof have been a topic throughout this article. In this section, I shine a light on some additional aspects.

Spoofing

As we saw [earlier](#), the sender of an email can easily be [spoofed](#) because at least historically emails aren't authenticated. Somewhat frustratingly, [RFC 5321](#) and [some companies](#) see forged sender addresses more as a feature than as a bug. Criminals abuse this "feature" to trick unsuspecting users into performing actions or disclosing information, which they wouldn't do otherwise. Exploiting the credulity of people is known as [social engineering](#). Besides impersonating a trusted organization for [phishing](#), a common attack is to send a victim an email which seemingly comes from their own address. In the message, the attacker claims that they've compromised the victim's computer and that they've recorded the victim masturbating to porn. The attacker threatens to send the recording to all the victim's contacts unless they receive a payment, usually in [Bitcoin](#), within a couple of days. This form of [blackmailing](#) is known as [sextortion](#). If you receive such an email yourself, how do you know that the attacker's claim is wrong? First of all, you know now that the sender address of emails can easily be forged and that there is no reason to assume that your account has been compromised. But more importantly, if there was an easy way to increase the fraction of people who pay the ransom, criminals would certainly make use of it. In the case of sextortion, they would just have to include a screenshot of the recording and the addresses of some contacts to make presumably the large majority of people pay. Given that this is (usually) not the case, there's no reason to worry. Do people fall for this crap? The answer is yes, unfortunately. The first time I received such a message was on 13 January 2019. The fraudster demanded 356 Euro in Bitcoin to remain silent and was stupid enough to provide the [same Bitcoin address](#) to several victims. Since all Bitcoin transactions are public, we know exactly how much money they made: 5.379 BTC, which was worth around 20'000 USD at the time. This also means that they had no way to know who of their victims actually paid, which made their threat even less credible to anyone who has a basic understanding of [blockchains](#).

Besides social engineering, spoofed sender addresses can be abused where emails are [used for authentication](#). For example, people can often unsubscribe from [mailing lists](#) via email. Even if this is not the case, many mailing lists remove subscribers to whom several messages in a row couldn't be delivered automatically. Unless a mailing list uses unpredictable [variable envelope return paths \(VERP\)](#), [bounce messages](#) can easily be forged, which means that you can unsubscribe other people from the mailing list. Similarly, it's often the case that only approved senders can send a message to all subscribers of a mailing list. Anyone who knows how to spoof emails can easily bypass this restriction and spam the mailing list.

Email address spoofing can be prevented by enforcing [domain authentication](#), which I'll cover in the last chapter of this article.

Phishing

Impersonating a trusted organization to obtain sensitive information or payments from gullible users is known as phishing. Phishing emails often direct their victims to a fraudulent website which looks exactly like the legitimate website. By providing a pretext, the attacker tries to get the victims to perform a specific action, such as entering their username and password or initiating a payment with their credit card. Phishing attacks can target specific individuals or a diverse group of people. If they're not just an advance-fee scam, they usually require some technical skills to execute them. This is why most phishing attacks are motivated by financial gain rather than a desire to harass or stalk the victim. While requesting a payment leads to a direct success for the criminals, usernames and passwords can be used to launch further attacks from the victim's account. For example, the credentials of an employee can be used to infiltrate a company in order to obtain trade secrets or to install ransomware on their computers.

Phishing attacks come in all shapes and sizes, but you can reduce your risk by sticking to the following principles:

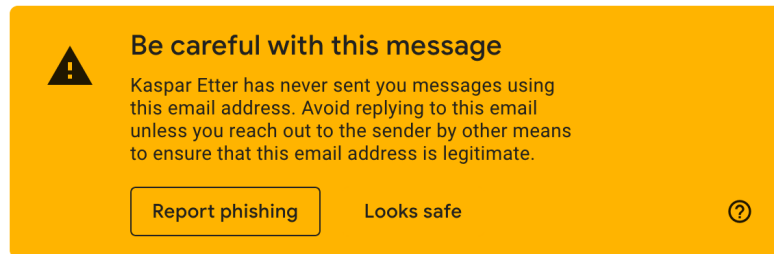
- **Always be suspicious:** If an email prompts you to perform a certain action, your alarm bells should ring. Have you been prompted for similar actions before? Is the time frame to perform the action unusually short? Is there a reasonable default option if you don't perform the action? Does the action involve the disclosure of sensitive information or a payment?
- **Don't click on links:** Phishing attacks require that you take the bait. Create a bookmark for all the websites where you have an account. Make it a habit to navigate to these websites yourself instead of following links. If an email says that a subscription is about to expire, log in to the website of the service provider with the bookmark and not the link. Using a bookmark (or a search engine) to navigate to a website is better than relying on the address autocompletion of your browser. If you clicked on a dubious link by mistake in the past, the fraudulent URL is still in your browser's history and you may not be able to recognize it as such.
- **Hover over links:** If you can't suppress your urge to click on a link, move your mouse over the link first and verify whether the status bar at the bottom of the window indeed displays the address you want to visit. You should always do this because the text of a link can be misleading. For example, www.google.com takes you to Bing, not Google. You should check the destination of a link before you click on it. If you check the destination of a link only in the opened browser window, you have already confirmed to the attacker that you click on links, and the visited website might have already infected your computer with malware. Unfortunately, link tracking can make it quite difficult to recognize whether the destination of a link is legitimate. Furthermore, not all companies prime their users to trust only a single domain. For example, PayPal, of all companies, directs their users to paypal-communication.com instead of paypal.com when informing them about changes to the general terms and conditions. Additionally, homograph attacks can make it difficult or even impossible to recognize that the target domain is not the legitimate one. This is one more reason why you shouldn't click on links in the first place. The only exception to this rule are links to articles on which you won't perform any actions. However, this means that you have to remember for each tab of your browser whether the address came from a trusted or an untrusted source. Anything you open on an untrusted page can also not be trusted. Some mail clients, such as Apple Mail, don't have a status bar and show the destination address in a tooltip instead. And yes, Apple Mail is smart enough to override any tooltips that a sender provided with the title attribute. I've tested this.
- **Use a password manager:** Seriously. Password managers not only allow you to have a long, randomly-generated password for each website, they also prevent you from entering them on the wrong websites. To be precise, you can still paste your passwords into any input fields you like. Password managers just won't do this for you if the domain is different. This is just one more level of defense, which is especially useful for innocuous-looking actions that don't trigger your alarm bells. For example, some websites require you to log in before you can unsubscribe from their newsletter, and such an email and login can be bogus, of course.
- **Verify the sender:** Who sent you the email? Since the sender of an email can (still) be spoofed, a trusted sender shouldn't lower your level of suspicion much. If the domain in the From address doesn't belong to the impersonated organization, though, you should almost certainly ignore and delete the message. Mail clients could do way more to protect their users from phishing attacks. For example, changing the policy for incoming emails from "allowed unless blocked" to "blocked unless allowed" would likely help a lot in shifting the mindset of users. Mail clients could also display the country of origin for each message, warn the user if the message isn't authenticated or if the clicked link leads to a domain which is different from the From address, etc.
- **Disable display names:** While spoofing sender addresses can be prevented by technical means, the sender can choose their display name at will. Since sender-chosen means attacker-chosen, users shouldn't be confronted with unverified display names. Unfortunately, all the mail clients I've checked handle this aspect so badly that I had to write a separate box on this topic.
- **Confirm out-of-band:** If a known sender asks you to perform an action which has far-reaching or irreversible consequences, contact the sender through a different communication channel and let them confirm the request before executing it. Obeying orders blindly is dangerous from a security perspective and subordinates should be trained and encouraged to question them.

▼ Malicious display names

In my opinion, the sender's display name should be used as the suggested name only when the recipient adds the sender's address to their address book. Since the sender can choose any display name they want, it shouldn't be displayed anywhere else. Unfortunately, I'm not aware of any mail client which handles the display name like this. Gmail, Outlook.com, Yahoo! Mail, Apple Mail, and Mozilla Thunderbird even show only the sender's display name without the sender's email address in the inbox view. Thunderbird and Apple Mail on iOS, however, do show the sender's email address in angle brackets if the sender's display name is an email address. In the other clients, the user cannot tell whether the displayed address is the sender's email address or the sender's display name. You can test this by entering something like "bob@example.com" <alice@example.org> in the From field of the ESMTP tool. Please note that you have to quote the display name if you use periods or the @ symbol in it. Once you open the message, most mail clients also display the sender's email address. In Apple Mail, you

have to disable “Use Smart Addresses” under “Viewing” in “Settings” for the client to even show you the sender’s email address. On iOS, [this option doesn’t even exist](#). If you want to see who actually sent you a message, you have to click on the sender’s display name. Since we humans tend to [confirm what we already think](#) rather than to question our initial assumptions and beliefs, the behavior of these mail clients seems reckless to me. Many users might not check the sender’s email address when they think that they already know who the sender is based on the message overview screen.

The very least that mail clients should do to prevent phishing attacks is to show a warning if a known display name is used by an unknown sender. I have seen this only in the Gmail web interface so far. For some reasons, I can no longer replicate this, though.

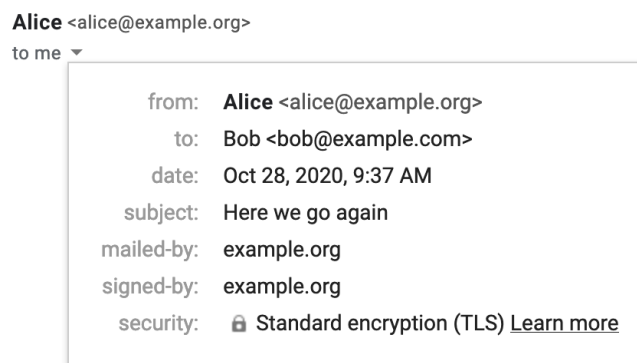


How Gmail warns its users when a known display name is used by an unknown sender.

Confidentiality and integrity

As we saw [earlier](#), the percentage of emails which are encrypted and authenticated in transit increased significantly over the last decade. When you send an email, though, there is no guarantee that the confidentiality and integrity of your message is protected when it is relayed from your outgoing mail server to the incoming mail server of the recipient. This is especially problematic when email is used to perform security-critical operations, such as [password resets](#). Due to backward compatibility, the [email protocols](#) are secure only against [passive attackers](#). I will cover the efforts to make email secure against active attackers in the [last chapter](#).

In my opinion, mail clients should warn their users if the incoming mail server of one of the recipients doesn’t support [strict transport security](#). You can increase the pressure on email service providers only by increasing the awareness of users. Gmail provides an easy way to see whether a received message has been authenticated and encrypted in transit, which allows users to assess the authenticity and, [somewhat misleadingly](#), the confidentiality of a message at least after it has been transmitted:



You can click on the little triangle to see more details in [Gmail’s web interface](#).
mailed-by indicates a successful [SPF check](#), signed-by a valid [DKIM signature](#).
security indicates that the outgoing mail server of the sender used [STARTTLS](#).

Reliable delivery (availability)

Besides confidentiality and integrity, [information security](#) is also concerned with the availability of a service. Since your message might be silently [discarded as spam](#) or land in the recipient’s spam folder, which they don’t check on a regular basis, you can never be certain that a (new) recipient received your message in their inbox. Most people minimize this risk by [not hosting their emails themselves](#). Once [domain authentication](#) is commonplace, which solves the problem of [backscatter](#), we can hopefully fight [spam](#) with [other techniques](#) so that self-hosting becomes feasible again.

Custom [email filters](#) are another source of unreliability. When users receive too many emails which they don’t want or can’t handle, they are tempted to set up a rule which moves or deletes them automatically. Personally, I have a rule which deletes all messages which contain certain keywords, such as “lotto winner”, in their subject. I’ve recently also added some [top-level domains](#), such as [.cheap](#) and [.city](#), to this list. If the From address ends in one of these domains, the message is deleted immediately. My custom anti-spam rule, which also includes the domains of

sales companies, does wonders for my inbox. The problem with custom email filters, though, is that they often work like shotguns: They certainly hit the messages you wanted to remove from your inbox, but due to their simplicity, they likely bring down legitimate messages as well. As long as senders send emails automatically, recipients will remove them automatically. Casualties are to be expected in such a setup.

Quoting HTML messages

HTML emails can be styled. When you reply to or forward such an email, your mail client has to make sure that the quoted message cannot change the appearance of your own message. If the quoted message isn't escaped properly, an attacker can inject text into the victim's response. When quoting an HTML message, mail clients need to ensure the following two things:

- **Scoped styles:** The style of the quoted message may not leak into the surrounding message. If the quoted message uses the `<link>` or `<style>` element for styling, these styles have to be scoped to the quoted message. Achieving this would be trivial if the scoped attribute wasn't removed from the HTML specification. If we're lucky, we might get an scope selector in a future CSS standard. Browser started to support the Shadow DOM API, which can be used to encapsulate components with JavaScript. Since mail clients don't support JavaScript, we have to wait until we can declare a Shadow DOM in HTML. So how do mail clients handle this? Gmail simply removes the `<style>` element when quoting an HTML message. If you want to make sure that your message is still displayed properly when it is replied to or forwarded, you have to keep inlining the styles. Apple Mail inlines internal CSS when quoting an HTML message. Yahoo Mail moves the `<style>` element from the `<head>` into the `<body>` and prefixes each rule with an ID, which it also assigns to the <div> element which contains the quoted message. Thunderbird only moves the `<style>` element from the `<head>` into the `<body>` and thus fails to scope the styles. Outlook.com behaves differently for replies and forwarded messages: It fails like Thunderbird in the former case and inline styles incorrectly in the latter case.
- **No overlays:** CSS can be used to move HTML elements away from their default position in a document. This becomes a problem when HTML elements in the quoted message can be moved above the attribution line since email users are trained to perceive everything above the attribution line as coming from the sender of the message. I can think of three ways how HTML elements can be moved around with CSS, but I wouldn't be surprised if CSS has more ways to achieve this. Firstly, there is the position property, with which elements can be moved relative to their default position or to an absolute position in the document. Secondly, the transform property can be used to translate HTML elements (and to scale and to rotate them). Thirdly, negative margins have a similar effect as position: `relative`; . Without having tested all possibilities, I have the impression that webmail clients handle this quite well. For example, Gmail doesn't list position and transform under supported CSS properties and also removes negative margins before displaying a message. Desktop clients, on the other hand, struggle with this. `position: absolute`; and `margin-top: -200px`; work in Apple Mail and in Thunderbird. Restricting styles to inline CSS isn't enough to scope the styles. Doing so would make it more difficult, though, to show the injected text only in the reply but not when composing the reply.

If analyzing the raw message before forwarding or replying to a message is too much to ask from you, you have only two options to avoid these issues: Choose a mail client which cares about your security or enforce that all messages are composed in plaintext. Apple Mail allows you to configure this in the "Composing" tab of your "Preferences": Change the "Message format" to "Plain text" and disable "Use the same message format as the original message". If you use Thunderbird, you can disable "Compose messages in HTML format" under the "Composition & Addressing" tab of your "Account Settings".

▼ Thunderbird example exploit

If you reply to or forward the following message with "No, I don't." in Thunderbird, the recipient will see "Yes, I do." instead. If you have already disabled the composition of messages in HTML, you have to press "Shift" when you click on the "Reply" or the "Forward" button. For this particular attack to work, the message has to be composed in the "Paragraph" style.

```
<html>
<head>
  <style type="text/css">
    p { font-size: 0; }
    p::before { content: 'Yes, I do.'; font-size: initial; }
    p + p { display: none; }
    p[_moz_dirty] { display: block; font-size: initial; }
    p[_moz_dirty]::before { display: none; }
  </style>
</head>
<body>
  <div>Hello, do you like my proposal?</div>
</body>
</html>
```

How to exploit Thunderbird's failure to scope the styles of the quoted message. [Click here](#) to use this example in the ESMTP tool above.
The attack uses the ::before pseudo-element to inject the text, and `p[_moz_dirty]` to hide the injected text during composition.

You can also use this attack to frame your boss so that it appears to someone in Cc as if the boss authorized a payment or some holiday. Unfortunately for the attacker, the content preview in the message list still shows “No, I don’t.”. Moreover, the exploit becomes apparent if the recipient inspects the raw message. While the attack isn’t perfect, the problem is certainly worrying and should be fixed. I reported [this vulnerability](#) to the Thunderbird team on 25 January 2021 and they decided to make the report public without a fix. There are some related issues with quoting HTML messages, which I’ll cover in the [next subsection](#).

▼ Outlook.com example exploit

If you reply to the following message with “No, I don’t.” on [Outlook.com](#), the recipient will see “Yes, I do.” on other clients.

```
<html>
<head>
  <style type="text/css">
    div[style] { display: none; }
    hr { border: 0; }
    hr:before { content: 'Yes, I do.'; display: block; border-bottom: 1px solid; }
  </style>
</head>
<body>
  <div>Hello, do you like my proposal?</div>
</body>
</html>
```

How to exploit Outlook.com’s failure to scope the styles of the quoted message. [Click here](#) to use this example in the [ESMTP tool](#) above. Since Outlook.com doesn’t copy styles like `div[style]:before` and `div:first-child:before` to the reply, I had to abuse the [<hr> element](#) to make the injected text appear only once.

If you click on “Forward”, Outlook.com inlines the CSS. However, it calls the inlining function on the quoted message wrapped with the attribution line instead of the original message. By adding the style `b { display: none !important; }`, the sender can hide the parts which are generated by Outlook.com, such as “From:”, “Sent:”, “To:”, and “Subject:”.

I reported both problems to Microsoft on 25 January 2021. The report was closed within a couple of hours, apparently because it lacked a valid proof of concept. I immediately objected to this assessment and got the following response from Microsoft two weeks later on 8 February 2021:

Thank you for your security research and submissions to Microsoft. The severity of this issue recently was reviewed and determined to be below the bar for further servicing as an MSRC case. At this time we are closing this case and will not provide further updates on this issue, however it may be addressed in a future version of our products.

Different appearances

Another issue with email is that the same message can appear differently to different recipients. This is a problem whenever you refer to the content of an earlier message, no matter whether you [quote the message](#) or reference it in the [In-Reply-To header field](#). Until mail clients address this issue, you must repeat the content you refer to. Emails can appear differently for three reasons:

- multipart/alternative: [Multipart messages](#) can include different versions of the same content so that the mail client of the recipient can display the last version whose content type it supports. However, nothing guarantees that the various parts contain the same content. [Spam filters](#) might flag messages whose alternative parts diverge too much from one another, but determining whether different parts contain the same content is more difficult than it seems. Let’s look at an example:

```
<html>
<body>
  Hi boss, can you confirm to our accountant in Cc that my monthly salary is
  increased by USD 100<span style="font-size: 0;">0</span> starting next month?
</body>
</html>
```

A simple HTML message whose plaintext version conveys a different content. [Click here](#) to use this example in the [ESMTP tool](#) above.

If your boss uses an HTML-capable mail client, they will see USD 100 in the message. When your boss replies to this message with “Yes, that’s what we agreed.”, all the mail clients I usually mention in this article generate a Content-Type: text/plain version of the reply, which includes USD 1000. If you know that your accountant uses a plaintext-only mail client, this attack will work. On most HTML-capable mail clients, you can see the plaintext version only by inspecting the [raw message](#). Thunderbird, however, allows you to change which part is being

displayed by switching the “Message Body As” in the “View” menu. If you use `display: none;` instead of `font-size: 0;`, Apple Mail won't include the additional zero in the plaintext reply as it uses something like `innerText` to determine the plaintext content. There are plenty of ways to hide content with CSS, though, and the plaintext conversion algorithm would have to consider them all. Since computing what is actually being displayed is impractical, the solution has to be to force all content to render in HTML messages by disabling those CSS properties. Since already the original message could have contained conflicting alternative parts, mail clients which take security seriously should probably warn their users when they reply to `multipart/alternative` messages because most mail clients hide the quoted messages in email conversations. All the mail clients I've tested generate the quoted message in the plaintext part of the reply from the HTML part that they've displayed to the user. If it wasn't for malicious CSS styles, mail clients wouldn't prepend your reply to content you haven't seen. The only problem that remains is that the `In-Reply-To` header field doesn't specify which alternative part your message refers to.

- **Conditional styles:** Even without alternative parts, the same message can be rendered differently on different devices due to media queries. The following message shows a different text on devices with a small screen than on devices with a large screen:

```
<html>
<head>
  <style type="text/css">
    @media (max-width: 599px) {
      .large { display: none; }
    }
    @media (min-width: 600px) {
      .small { display: none; }
    }
    .touch { display: none; }
    @media (pointer: coarse) {
      .touch { display: inline; }
    }
  </style>
</head>
<body>
  <p>
    You have a
    <span class="large">large</span>
    <span class="small">small</span>
    <span class="touch">touch</span>
    screen.
  </p>
</body>
</html>
```

A simple HTML message which is displayed differently on different devices. [Click here](#) to use this example in the [ESMTP tool](#) above.

Media queries are useful to design websites for various screen sizes, which is known as responsive web design. Since emails are read on a wide variety of devices, media queries are an important technique to make them look good on all devices. Since media queries and selectors aren't allowed in the style attribute, conditional rendering is much easier in mail clients which support internal or external CSS, which is the vast majority by now. In order to prevent this attack, Thunderbird no longer supports media queries. In my opinion, this is the wrong approach and the fix should rather be to force all content to render. Styles should affect only how content is displayed, not which content is being displayed. The supported media features vary greatly among clients. For example, the screen width media queries are supported by Gmail, Outlook.com, Yahoo Mail, and Apple Mail (also on iOS). The pointer media query, which can be used to detect a touch screen, is removed by the Gmail and Yahoo Mail webclients.

- **Different implementations:** As long as different users use different mail clients which sanitize emails differently, attackers can draft messages which are displayed differently to different recipients. Since it's easy to learn which mail client someone uses, it's often not difficult to have some part of a message be shown or hidden for a specific recipient. I've drafted such a message for you:

```
<html>
<head>
  <style type="text/css">
    .apple-mail, .outlook { display: none; }
    @media (pointer) {
      .apple-mail { display: inline; }
      div .apple-mail { display: none; }
      div .outlook { display: inline; }
    }
    @media (min-width: 0px) {
      .thunderbird { display: none; }
    }
  </style>
</head>
<body>
  <p>
    You have a
    <span class="large">large</span>
    <span class="small">small</span>
    <span class="touch">touch</span>
    screen.
  </p>
</body>
</html>
```

```

p:first-child .gmail { display: none; }
.yahoo-mail { display: none; }
p:first-child .yahoo-mail { display: inline; }
body .yahoo-mail { display: none !important; }
</style>
</head>
<body>
<p>
You're reading this message
<span class="apple-mail">in Apple Mail.</span>
<span class="thunderbird">in Thunderbird.</span>
<span class="gmail">on mail.google.com.</span>
<span class="outlook">on outlook.live.com.</span>
<span class="yahoo-mail">on mail.yahoo.com.</span>
</p>
</body>
</html>

```

A simple HTML message which is displayed differently by different clients. [Click here](#) to use this example in the [ESMTP tool](#) above.

As long as not all mail clients prevent senders from hiding content with CSS, email styling can be abused. Don't we have the same problem with websites? In principle, yes, but the difference lies in the expectation of users. On the Web, you know that pages are often customized and that their content can change at any moment. In the case of email, however, you expect that everyone sees the same content, especially when you quote another message. If you reply to messages without quoting them, an attacker can deliver a different message with the same Message-ID to each of the recipients. As I wrote earlier: Just because someone is listed as another recipient doesn't mean that they received the same message as you. The abuse of conditional CSS rules as a signing oracle was discovered and published by [Jens Müller and his colleagues](#) in 2019. The problem with diverging multipart/alternative parts was discussed thereafter in [this Thunderbird issue](#).

▼ Hide content with CSS

There are plenty of ways to hide text and other content with CSS. While this is useful on the Web, where you can have dynamic content, there is no reason to allow hidden content in emails, where you can't unhide it with JavaScript. (I know that one can accomplish amazing things with only CSS, such as tabbed areas, but do we really need this in emails?) Jens Müller and his co-authors included the following table of content-hiding CSS properties in [their paper](#), which I simplified and extended for you:

Property	Value(s)
<u>display</u>	none
<u>visibility</u>	hidden or collapse
<u>font[-size]</u>	0 [Helvetica] (also when combined with a distance unit or the percentage sign)
<u>color</u>	transparent, rgba(0,0,0,0), hsla(0,0%,0%,0) (for all RGB and HSL values)
<u>background[-color]</u>	(when used to match the color of the text)
<u>opacity</u>	0
<u>filter</u>	opacity(0), opacity(0%), or brightness(100)
<u>clip[-path]</u>	circle(0) (and other shapes that don't overlap the content)
<u>margin</u>	0 0 0 -1000px (also for the individual side properties)
<u>position</u>	absolute or relative with left: -1000px
<u>transform</u>	translateX(-1000px) or scale(0)
<u>[text-]overflow</u>	hidden (combined with [max-]width or [max-]height)

The many ways to hide text and images with CSS. If you can think of another way, [let me know](#).

For some of the properties, such as font-size and opacity, mail clients should enforce a minimum value. For other properties, such as display, only certain values should be allowed. Many of the properties can be removed completely. For example, [Gmail doesn't support](#) visibility, filter, clip[-path], position, and transform at all. Interestingly, the [text-]overflow property can be used to hide content based on the screen size of the client even if only inline styles are allowed. In order to make a static analysis even possible, all [CSS functions](#), such as [calc\(\)](#), should be removed. Otherwise, an attacker can simply replace opacity: 0 with opacity: calc(1 - 1).

Complexity

Since you made it to this paragraph, I probably don't need to convince you that email is incredibly complex. Email is a system that has been retrofitted to modern requirements for 40 years. It's no wonder then that what we have today is a complicated patchwork of extensions. Just to be clear: I don't want to criticize anyone in this section. Most of the design decisions that led us to the current situation were reasonable at the time. I still think it's a good idea to assess what brought us here as this allows us to appreciate what we have now. In my view, the following limitations of early email are responsible for most of today's complexity:

- **Text-based protocols:** Using characters to delimit the various parts of protocols and messages makes it easy for us to interact with servers manually, but it also prevents us from sending arbitrary content without escaping it first. SMTP and POP3 require periods to be escaped, IMAP, Sieve, and ManageSieve require user-provided arguments to be escaped, and multipart messages require unique boundaries. None of this is tragic but conversions are always a potential source of errors and incompatibilities.
- **Line-length limit:** Since each line of a message may consist of at most 1'000 characters, folding whitespace is required for header fields, long text lines have to be broken without conflicting with quoting conventions, and system-specific newline characters must be converted to {CR}{LF}.
- **ASCII-only characters:** Email is older than ISO 8859 and Unicode. To remain backward compatible, non-ASCII characters have to be encoded in the message body, in header fields, in domain names, in parameter values, and in URLs. To make things even more complicated, all these encodings are different. When the involved servers support SMTPUTF8, UTF-8 can be used in the local part of an email address, but internationalized messages have to be downgraded for clients which don't support UTF-8.
- **No submission protocol:** In the early years of email, mail clients could submit outgoing messages to any mail server without authentication. As a consequence, mail submission and mail retrieval were handled completely differently. In order to make the change for existing mail clients as small as possible, the mail submission protocol was forked from ESMTP rather than being incorporated into access protocols, such as POP3 and IMAP. Unless their email service provider is in a configuration database, users have to configure both their incoming mail server and their outgoing mail server to this day. If they change their passwords, they usually have to enter it twice in their settings. Furthermore, it can happen that they can receive messages but cannot send them and vice versa. For ordinary users, this is really confusing. This distinction between incoming mail server and outgoing mail server is also the reason why messages have to be submitted twice if you want to record the sent messages in your mailbox. After two decades of little progress with regard to access protocols, IMAP finally addresses this and many other issues.
- **No transport security:** Emails and account passwords couldn't be secured in transit for more than a decade. Once Transport Layer Security (TLS) became popular, existing protocols were retrofitted so that all communication could be encrypted and authenticated. All the protocols were extended to support Explicit TLS, but all of them require different commands to activate TLS, which makes it difficult to use some of them from the command line. The introduction of protocol variants which use Implicit TLS required additional port numbers, which confuses ordinary users even more. Since mail servers don't know whether other mail servers support TLS, the communication between them is still vulnerable to downgrade attacks. I'll cover in the last chapter of this article how such attacks can be prevented.
- **No sender authentication:** Since emails aren't authenticated, it's quite easy to spoof the sender of a message. This aggravates problems such as spam and phishing, and it can lead to undesirable backscatter. I'll explain the mechanisms which are used to alleviate this issue in the last chapter.

▼ Benign inconsistencies

As we have seen in the previous section, complexity is bad for security. As we will see in the next section, complexity is also bad for innovation. But for now, let's have some fun with a few benign inconsistencies, which are caused by email's complexity:

- What should happen when a user enters a Subject which is already Encoded-Word encoded? In my opinion, the recipient should see exactly what the sender entered. In other words, `decode(encode(Subject)) = Subject` should always hold even if the Subject is already encoded. (Note that `encode(decode(Encoded)) = Encoded` is not the case for encodings in which more characters can be escaped than necessary.) As far as I can tell, RFC 2047 doesn't specify how mail clients should handle this. It just says at the bottom of page 8: "In rare cases it may be necessary to encode ordinary text that looks like an Encoded-Word." When you paste `=?ISO-8859-1?Q?A1Buenos_d=EDas!?=`, which is the Encoded-Word encoding of `iBuenos días!`, into the Subject field of various mail clients, they send the following Subject header field:

Mail client	Subject encoding
Apple Mail	<code>=?us-ascii?B?PT9JU08t0Dg10S0xP1E/PUExQnVlbm9zX2Q9RURhcyE/PQ==?=</code>
Thunderbird	<code>=?UTF-8?B?qFCdWVub3MgZM0tYXMh?= =</code>
Gmail	<code>=?UTF-8?B?qFCdWVub3MgZM0tYXMh?= =</code>
Outlook.com	<code>=?iso-8859-1?Q?A1Buenos_d=EDas!?=</code>
Yahoo! Mail	<code>=?ISO-8859-1?Q?A1Buenos_d=EDas!?=</code>

How various mail clients encode the subject `=?ISO-8859-1?Q?A1Buenos_d=EDas!?=`.

Only Apple Mail produces an encoding in which the recipient sees what the sender entered. Thunderbird and Gmail recognize that the user entered an Encoded-Word, but instead of escaping it, they decode and re-encode the user-provided string. Outlook.com also recognizes that the user entered an Encoded-Word but then only lowercases the character set. 🤖

- What the standard does say is that compliant mail clients must ensure that all words which begin with =? and end with ?= are valid Encoded-Words. So what happens if you enter =?He11o?= in the subject line? Only Apple Mail encodes this as =?us-ascii?B?PT9IZWxsby89?=. Thunderbird, Gmail, Outlook.com, and Yahoo Mail leave the string as is and thus don't conform to RFC 2047.
- RFC 5322 states that runs of whitespace are to be interpreted as a single space character in structured header fields. The Subject, however, is an unstructured header field, where only newline characters which are followed by whitespace are to be removed. Gmail, Outlook.com, and Yahoo Mail display adjacent spaces in the Subject as a single space. Thunderbird is a bit of a special case: It displays adjacent spaces in the list view but not in the detail view of a single message. Consistency is not even achieved within the same mail client. Once more, only Apple Mail conforms to the standard.

If you're aware of other inconsistencies, let me know.

▼ Unreasonable decisions

I wrote above that most of the design decisions which led us to the current situation were reasonable at the time. As you might have noticed, "most" is not "all". So here comes the list of things which should never have been approved or implemented:

- Comments in header fields: Comments can appear almost anywhere in structured header fields. Comments are wrapped in parentheses and comments can be nested. It should have been clear from the very beginning that users won't compose and parse RFC 5322 messages themselves. Since emails are almost always parsed by mail clients, which ignore any comments, comments increase the complexity of the message format without bringing any benefits. Where comments do have some merit, such as in the Received header field, an extensible list of name-value pairs similar to the parameters in MIME header fields could have been used instead. You find an example message with plenty of comments in Appendix A.5 of RFC 5322. (The example violates a SHOULD NOT two times, though.)
- Address syntax: Determining whether an email address is valid or whether two addresses are the same should be easy. I think I have a good understanding of the former question but I still struggle with the latter. Given that arbitrary strings can be encoded with only letters and digits if needed, the syntax of email addresses should never have been so complicated. I believe it's a perfect example of where less would have been more.
- Vague Bcc semantics: RFC 5322 requires only that Bcc recipients are never disclosed to non-Bcc recipients. Within this constraint, implementations can do pretty much anything they want. In my opinion, user-facing behavior should be fully specified because users cannot be expected to study how a particular email setup behaves.
- MIME version: The MIME-Version: 1.0 header field is pointless, which is also acknowledged by the author himself.
- HTML emails: In my opinion, HTML emails were introduced prematurely without thinking through the implications first. At the very least, a subset of HTML and CSS should have been standardized, which all compliant mail clients have to support. If this was the case, we might have a flag to disable undesirable CSS properties directly in the rendering engine by now. Additionally, such an effort would likely also have led to a reasonable way to scope CSS.
- Remote content: Mail clients should never have supported remote content as it violates fundamental principles of email.
- Domain lookup: Falling back to A and AAAA records if a recipient domain has no MX records causes problems without bringing any substantial benefits.

If you want to have something added to or removed from this list, let me know.

Innovation

Besides IMAP, dynamic content, and what we'll discuss in the last chapter, there was barely any innovation over the last two decades. This is a pity given that email is the only decentralized communication service with global adoption. I can only speculate about the reasons for the lack of innovation:

- **Complexity**: The enormous complexity of email can deter software engineers from entering the field. Patching a heavily patched system further is also not appealing to many young talents. I hope this article can motivate more people to shape the future of email in a positive way.
- **Fragmentation**: The email ecosystem is so fragmented that no single organization can push the industry forward. The innovation that we see, such as email markup and dynamic content, often remains limited to just a few companies. If you want to write a mail client for a general audience, you have to support IMAP. If you have to deal with the intricacies of IMAP anyway, you don't gain anything by implementing a newer access protocol such as IMAP as well. As long as all mail clients which people want to use support IMAP, existing email service providers have little incentive to support IMAP.
- **Saturation**: The email market is saturated with free solutions for clients, servers, and hosting. The low willingness to pay for a product or service makes it really hard to build an innovative business in this space. Combined with the inertia of users, there is almost no economic

pressure to innovate. Email service providers with a strong focus on privacy are the only exception to this rule because more and more people realize that if they don't pay for a service, they're the product and not the customer.

Format innovation

Since Skype failed to innovate, it was superseded by Zoom. The same fate is happening to WhatsApp: Telegram is showing us how much room for innovation there is for a messaging app. There's plenty of features I would like to see in email. For a start, we still have no No-Reply header field, no Proof-Of-Work header field, no header field to reference the previous message by its hash (ideally using a Merkle tree for MIME parts so that attachments can be removed from a message without invalidating its hash), no header fields for the sender's contact details to replace email signatures, no content type to initiate and reply to surveys, etc.

Some features, such as message compression, exist in theory but not in practice. Other features, which originated in the alternative email system X.400, were formally specified as IETF email header fields in order to increase compatibility between the two systems but were never recommended for general use. Among these header fields are Supersedes to replace a sent message with a revised version, Expires to indicate when a message loses its validity, and Reply-By to request a response in the specified time period.

Client innovation

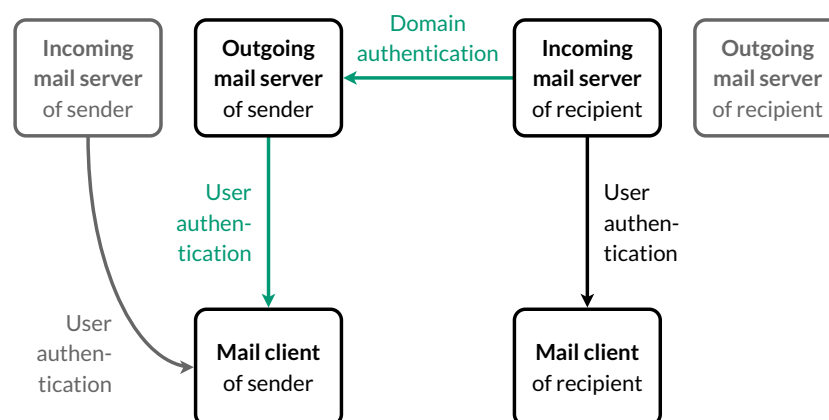
Given the decentralized nature of email, protocol and format innovations are difficult to achieve. However, nothing hinders mail clients from innovating at the edge of the network. I've mentioned plenty of ideas throughout this article. Among them are sender approval, automatic challenges, Bcc recovery, privacy features such as proxying remote content via Tor (and even submitting emails via Tor as long as email service providers leak the IP addresses of their users), and security features such as preventing malicious display names and different appearances of messages. It would be great if my mail client displayed whether a received message was successfully authenticated with SPE and DKIM (just like Gmail). I would like to see native support for DNS-based autoconfiguration, Sieve and ManageSieve, as well as PGP. I don't understand why mail clients separate the outbox from the inbox. (I don't know any other messaging app which does this, and just because IMAP uses folders doesn't mean you have to display them.) I think it would be great if my mail client could timestamp all the emails that I send. Whenever I submit a responsible disclosure, I do this manually.

Fixes

The last chapter of this article is dedicated to recent standards which address some of the aforementioned security issues. We'll study how spoofing is prevented with domain authentication and how confidentiality and integrity is ensured in the presence of an active attacker with strict transport security. Many of the approaches rely on the Domain Name System (DNS) to provide additional information. This is secure only if the records are authenticated with DNSSEC. I will no longer mention this aspect in the remaining subsections. Some of the steps have to be performed by the owner of the domain rather than the email service provider. If you use a custom domain for your emails, you should definitely read the part about domain authentication to make sure that your domain is configured properly. Since email is a decentralized service, we can improve its security only in a collective effort.

Domain authentication

Historically, the sender of an email was not authenticated: Anyone could relay a message to anyone using any From address they wanted. Impersonating another sender is known as spoofing. While the prevention of spoofing won't eliminate spam and phishing on its own because spammers can implement the following standards as well and phishing remains possible with similar domains and malicious display names, it's an important prerequisite for other techniques, such as flagging unknown senders. As we've seen earlier, email spoofing is addressed in two steps: The incoming mail server of the recipient verifies that the other party is authorized to send emails on behalf of the sender's domain and the outgoing mail server of this domain ensures that the local part of the From address belongs to the user who submitted the message.



The incoming mail server of the recipient authenticates the outgoing mail server of the sender and the outgoing mail server of the sender authenticates the user who submits the message.

As the title suggests, this subsection is only about the first part of the problem, namely how a domain owner can specify which mail servers are authorized to send messages on behalf of the domain and how receiving mail servers can verify whether the sending mail server is indeed authorized for the claimed domain. The second part is usually solved with password-based authentication mechanisms. The following techniques don't prevent spoofing if the outgoing mail server of the sender is compromised or if the attacker can create an account at the same email service provider and impersonate another user during submission.

Before you continue, make sure that you understand the difference between a message and its envelope.

There are three complementary standards for domain authentication:

- **Sender Policy Framework (SPF)**: List the IP addresses of your outgoing mail servers in a DNS record at your domain. SPF protects only the MAIL FROM address, which is used for bounce messages. SPF can cause problems with email forwarding.
- **DomainKeys Identified Mail (DKIM)**: Let the outgoing mail servers sign outgoing messages and publish the public key in a DNS record at the sender's domain. The signature usually survives email forwarding but introduces non-repudiation.
- **Domain-based Message Authentication, Reporting and Conformance (DMARC)**: Publish a policy, which tells recipients what to do with messages that fail both SPF and DKIM, in a DNS record at your domain. Without a DMARC record, the recipient cannot know whether the sender uses DKIM. By publishing a DMARC policy, you also require the domain in the From address to match the domain in the MAIL FROM address and the DKIM header field. Moreover, you can specify an email address to which receiving mail servers can send aggregate reports so that you know how your DMARC policy affects the delivery of your own messages.

▼ Adoption benefits

At first glance, it seems as if configuring your domain with the above standards benefits mostly others. Why should you invest some of your valuable time in your email setup just to protect others? In economics, benefitting unrelated third parties without being compensated for it is known as a positive externality. Treating information security as a public good, one might expect to see many free riders, who benefit from improved security without contributing to it. Fortunately, this is not what is happening with the above standards as they benefit the people who deploy them on their domains as well:

- **Deliverability**: Protecting your domain with SPF, DKIM, and DMARC records makes it more difficult for others to abuse your domain for spamming and phishing. When fewer messages coming from your domain are marked as spam, the reputation of your domain improves, which increases the chance that your messages reach their recipients.
- **Flexibility**: By specifying which servers are authorized to deliver email on behalf of your domain, you make it possible for others to attach reputation to your domain rather than to the IP addresses of your outgoing mail servers. This allows you to change your servers and deploy additional ones without losing the reputation that you've built so far.

Unfortunately, these benefits apply only to the domains which you use to send emails from. From a security perspective, however, it's just as important to configure SPF and DMARC records for the domains which you don't use to send emails from. Since the above incentives exist only for the former category of domains but not the latter, mail server authorization is widely deployed on primary domains but less so on redirect domains.

▼ Domain owner

Knowing with certainty that a message was sent from a specific domain is important for algorithms such as reputation systems and email filters, but domain authentication can give human users a false sense of security, which is dangerous. On the one hand, it can be difficult for us to tell different domain names apart, i.e. we easily fall victim to homograph attacks. On the other hand, it can be really hard to figure out who owns the domain in question. There are around 1'500 top-level domains and you likely don't know which top-level domain each of your contact uses. Do they use a generic top-level domain or a country-code top-level domain? Is the second-level domain only the company name or did they have to prefix it with something because the other name was already taken? How can you determine whether a domain indeed belongs to the company you think it belongs to? Unfortunately, there's no simple answer to this question. The easiest thing you can do is to search for the company with your favorite search engine. If the domain in the email matches their web presence, you're done. If a company doesn't use a single domain across all channels (such as myself with this blog), the best you can do is to query the WHOIS database.

WHOIS is a simple protocol to query information about registered domain names. It is specified in RFC 3912 and is typically used with the whois command-line utility. If you want to look up the information for ef1p.com, you enter `whois ef1p.com` into your command-line interface. The whois utility opens a TCP connection on port 43 to various WHOIS servers. IANA maintains a root WHOIS server at `whois.iana.org`, which you can query to determine the WHOIS server of the registry, which in turn informs you about the WHOIS server of the registrar. For example, this is how you find the WHOIS information of ef1p.com:

```
$ telnet whois.iana.org 43
com
[...]
whois: whois.verisign-grs.com
```



```
[...]
Connection closed by foreign host.
$ telnet whois.verisign-grs.com 43
eflp.com
[...]
Registrar WHOIS Server: whois.gandi.net
[...]
Connection closed by foreign host.
$ telnet whois.gandi.net 43
eflp.com
[...]
Registrant Name: REDACTED FOR PRIVACY
[...]
Connection closed by foreign host.
```

How to perform WHOIS queries yourself. Once the TCP connection is established, you just enter the domain name of interest followed by a new line. (Telnet should convert "return" into {CR},{LF}, automatically.) As soon as the server has sent the answer, it closes the connection.

If you perform the above steps, you'll find that almost all information is redacted for privacy. This practice has become the norm rather than the exception and there is a proposal to abolish the WHOIS system as we know it altogether. And even if you do get a useful answer, the provided information might not be trustworthy since domain registrars don't verify their customers.

You might be tempted to simply enter the domain in a web browser but this is dangerous because you might get infected with malware. You also don't learn anything by doing so: Any website can look like the legitimate website and any website can redirect your browser to the legitimate website. The only way to be quite certain that a domain belongs to a given company is when the connection is secured with an extended-validation certificate before any redirects occur. Unfortunately, domain-validated certificates are much more common nowadays and many browsers hide the name of the owner even if this information is present in the certificate. In summary, you should be vigilant even if the domain looks trustworthy.

▼ Privacy implications

Similar to remote content, domain authentication triggers requests from the receiver. As a sender, the company which runs the name servers of your domain might learn the domains to which you send emails. As a recipient, the sender might learn the domains to which you forward incoming messages. Thus, email leaves more traces with domain authentication than without.

▼ Name chaining attacks

Remote content and domain authentication allow an attacker to trigger DNS lookups to their name server. If the DNS resolver of the victim isn't careful in how it processes the response, the attacker can poison its cache. If successful, completely unrelated requests from the victim go to the attacker instead of the intended recipient. For example, the attacker could reply to a query for the A record of `subdomain.attacker.example` with a CNAME record pointing to `target.example` and tell the victim's DNS resolver in the so-called additional section of the response that the A record of `target.example` has the attacker's IP address as its value. The attacker could also redirect the victim's DNS resolver to the name server of `target.example` and provide their own IP address as the name server's address in the additional section of the response. This is known as a name chaining attack and in the absence of DNSSEC, any records in the additional section may not be cached by DNS resolvers. (In case you are wondering, example is a top-level domain reserved for examples.)

Sender Policy Framework (SPF)

The Sender Policy Framework (SPF) is specified in RFC 7208. As a domain owner, you list the IP addresses of your outgoing mail servers in a TXT record at your domain. Incoming mail servers then check whether the IP address of the sending mail server is included in the SPF record of the domain which was used in the MAIL FROM command. If this is not the case, incoming mail servers can reject the message during the SMTP session. RFC 7372 defines enhanced status codes with which the server can indicate failed SPF validation. Since IP addresses cannot be spoofed without access to the target's local network, this procedure authenticates the sender's domain.

So how do you create the SPF record for your domain? If you don't run your email server yourself, your SPF record will consist of:

- **Version:** Every SPF record has to start with `v=spf1`.
- **Includes:** An SPF record can include the IP addresses of another SPF record. Search for the appropriate record from your email service provider. For example, put `include:_spf.google.com` (source) into your SPF record if you use Google Workspace. Since mailing list providers such as Mailchimp use an address of their own in the MAIL FROM command so that they can handle bounce messages for you, you don't need to add the addresses of their servers to your SPF record.

- **Default:** Provide an explicit default result for any sender which didn't match one of the previous mechanisms. If you want incoming mail servers to reject messages with a spoofed MAIL FROM domain, use `-all`. If you want incoming mail servers to just flag such messages as potentially fraudulent, use `~all`. In order not to disrupt email forwarding, incoming mail servers are unlikely to enforce your SPF policy. They are much more likely to enforce the domain policy of your DMARC record.

An SPF record created according to the above steps looks as follows: `v=spf1 include:_spf.google.com -all`. On domains from which you don't send any emails, you should use `v=spf1 -all`. The full syntax of SPF records is much more powerful than this but rarely needed. I will cover SPF in more detail in the boxes below. There are a lot of things that can go wrong when configuring an SPF record. For a start, a domain may have at most one SPF record and the number of additional DNS lookups an SPF record may trigger is limited. Instead of listing all the pitfalls here, I've built a tool which performs 30 different checks on your SPF record. It uses Google's DNS API to query the records. Please note that this tool warns you only about common mistakes, it doesn't verify whether your outgoing mail servers are included in the record. You still have to test your setup by sending emails and checking the Received-SPF header field. By not evaluating whether an IP address passes SPF validation, the tool is also limited in other regards.

Domain:

Query



▼ SPF-Received header field

Since DNS records can change over time and the SPF check has to succeed only at the point of delivery, incoming mail servers should record the result of their SPF evaluation in the header of accepted messages. This allows mail filters and mail clients to process and display messages differently depending on whether the domain of the sender was successfully authenticated. There are two fields for this purpose: Received-SPF and Authentication-Results, which we'll discuss later. Both of them are trace fields, which means that they should be added at the top of the header and that they must appear above all fields with the same name. The format of the Received-SPF header field is as follows:

```
Received-SPF: {Result} ({Comment})[ {Key}={Value};]*
```

The format of the Received-SPF as specified in RFC 7208. The curly brackets need to be replaced with actual values. The content in square brackets is optional and the asterisk indicates that the preceding content can be repeated.

The possible results of SPF evaluation are pass, fail, softfail, neutral, none, temporary error, and permanent error. The first four values are the result of a successful evaluation (see the qualifiers and mechanisms for more information), none means that the sender used a syntactically invalid domain or that no SPF record was found, temporary error means that a temporary error occurred and that a later retry might be successful, and permanent error means that the error is permanent and needs to be resolved by the publisher of the SPF record. The Comment and the Key-Value pairs are recommended but optional. Typical keys can be taken from the example below:

```
Received-SPF: pass (example.org: domain of example.com designates 192.0.2.1 as permitted sender)
client-ip=192.0.2.1; envelope-from="sender@example.com"; helo=server.example.com;
```

An example Received-SPF header field. The values are intended to make the result verifiable.

▼ Protecting subdomains

Incoming mail servers query only the domain of the MAIL FROM address (or the HELO identity as a fallback) for an SPF record. If `support.example.com` doesn't have an SPF record, SPF verifiers won't continue the lookup with `example.com`. Does this mean that you should configure an SPF record for all your subdomains? If you use a subdomain to send emails from, then the answer is yes. If you don't use the subdomain for outgoing emails, the answer is more complicated. If the subdomain has an MX record to receive incoming emails, you should consider adding an `v=spf1 -all` record. Since many mail servers check whether the sender's domain has a valid MX record, configuring an SPF record on subdomains without an MX record is usually not necessary. For unused domains, setting up a DMARC record is much more important. On the one hand, DMARC protects the identity which is actually visible to users: the From address. On the other hand, DMARC policies also cover subdomains, where the policy for subdomains can be different from the policy for the organizational domain. If you do configure a DMARC record for unused domains, an SPF record is necessary only for mail servers that verify SPF but don't support DMARC. Given that SPF was introduced in 2006 and DMARC in 2015, such servers exist and won't vanish anytime soon.

Subdomains pose another interesting question: Are SPF verifiers supposed to follow CNAME records? If the answer is yes and an additional DNS query is necessary, does this count towards the lookup limit? Unfortunately, the standard is silent on this. The above tool does resolve CNAME records but doesn't count them. In the absence of an authoritative answer, I advise against using CNAME records. If SPF verifiers are supposed to follow CNAME records, then any CNAME record which points to another DNS zone is also a security risk because the owner of that zone can configure an SPF record and send emails on behalf of your subdomain.

▼ Email forwarding

Historically, emails to alias addresses were forwarded without changing the MAIL FROM address, which had the advantage that bounce messages were sent directly back to the original sender. Nowadays, forwarding emails without changing the MAIL FROM address breaks SPF validation. The incoming mail server of the final recipient might reject the message because the forwarding mail server isn't authorized to send messages on behalf of the sender's domain. In order to avoid this, forwarding mail servers have to use a sender rewriting scheme, which makes use of variable envelope return paths (VERP). The idea is to encode the original MAIL FROM address in a new MAIL FROM address so that bounce messages coming back to this address can be forwarded to the original sender. In order not to forward spam to the original sender, the forwarding mail server should use bounce address tag validation (BATV). In practice, most mail servers also accept emails with failed SPF checks if the reputation of the forwarder is high enough. As a consequence, forwarding usually works even without rewriting the MAIL FROM address, which defeats the purpose of SPF to some degree. [RFC 7208](#) also discusses some other approaches to this problem.

▼ SPF qualifiers

SPF records are structured according to the following syntax as specified in [RFC 7208](#):

```
v=spf1 [ [{Qualifier}]{Mechanism}]* [ {Modifier}]*
```

The curly brackets need to be replaced with a qualifier, a mechanism or a modifier. The content in square brackets is optional. The asterisk indicates that the preceding content can be repeated.

SPF records are evaluated from left to right. The qualifier of the first mechanism which matches the sender's IP address determines the result of the evaluation. If no mechanism matches, the default result is neutral. The four qualifiers are:

Qualifier	Result	Description
+	pass	The sender is authorized to send messages on behalf of the given domain.
-	fail	The sender is not authorized to send messages on behalf of the given domain.
?	neutral	The domain owner makes no assertion. This result has to be treated like none.
~	softfail	Between fail and neutral. The message can be flagged but not rejected.

The four qualifiers and the evaluation results to which they lead.

Mechanisms without a qualifier default to +. How the various results are to be handled is discussed in [section 8 of RFC 7208](#).

▼ SPF mechanisms

[RFC 7208](#) defines eight mechanisms to match the sender's IP address:

- all matches all IP addresses and is used to provide an explicit default result as the rightmost mechanism.
- ip4 and ip6 produce a match if the sender's IP address is in the network specified after a colon. The network is written in the classless inter-domain routing (CIDR) notation: An IP address followed by the number of bits that need to match. For example, ip4:192.0.2.0/24 matches all IP addresses from 192.0.2.0 to 192.0.2.255. When specifying a single address, /32 doesn't have to be appended to the address. The number in ip4 and ip6 refer to the version of the Internet Protocol.
- a matches if the domain has an A or AAAA record which matches the sender's IP address. When used without an argument, the lookup is performed on the domain of the SPF record. At the beginning of the SPF check, this is the domain of the MAIL FROM address. When additional lookups are triggered, the domain of the current lookup is used. Instead of using the current domain for the record lookup, a different domain can be provided after a colon. For example, a:example.com matches if the sender's IP address is among the A or AAAA records of example.com. The address range can be extended with a CIDR suffix.
- mx matches if the sender's IP address belongs to one of the mail exchange (MX) hosts of the domain. Since MX records contain a domain name and not an IP address, an additional lookup is required for each returned MX record. These lookups count towards the limit of 10 DNS queries after the initial record retrieval. Therefore, use the mx mechanism only if absolutely necessary. A different domain can be specified after a colon and the address range can be extended with a CIDR suffix.
- include matches if the evaluation of the referenced SPF record results in a pass for the sender's IP address. This is the most confusing of all mechanisms because the referenced SPF record is not actually included. If the evaluation of the referenced SPF record results in a pass, the qualifier in front of the include mechanism determines the result of the evaluation. Mechanisms with a -, ?, or ~ in front of them have no effect in the referenced SPF record unless they prevent a later mechanism from producing a pass result. For example, if the SPF record of example.com contains -include:example.net and the SPF record at example.net is v=spf1 -a ip4:192.0.2.1 -all, the -a

matters only if it produces a fail for the IP address 192.0.2.1. If this is not the case and the sender's IP address is 192.0.2.1, the evaluation of the referenced SPF record results in a pass. This causes the include mechanism of example.com to match and since the qualifier of the include mechanism is -, the evaluation of example.com's SPF record results in a fail. Records with such exclusions are quite rare but being able to exclude some addresses before authorizing other addresses can be useful when a range of IP addresses shall be authorized with a few exceptions.

- exists matches if the domain name provided after the colon has an A record. This is useful only in combination with macros, where parts of the provided domain name can be replaced with the local part of the MAIL FROM address or the sender's IP address. This makes it possible to produce different results for different users.
- ptr matches if the reverse DNS entry of the sender's IP address returns a subdomain of the target domain. The target domain can be provided after a colon similar to the a and mx mechanisms. If no such domain is provided, the target domain is the current domain of the SPF evaluation. Since the sender controls the reverse DNS entry of its IP address, the returned domain name has to include the sender's IP address in one of its A or AAAA records. This is known as forward-confirmed reverse DNS. According to RFC 7208, the ptr mechanism should no longer be used because it is slow and places a large burden on the .arpa DNS servers. As an errata points out, both arguments are questionable since most mail servers do a reverse lookup anyway to record the sender's identity in the Received header field.

Mechanisms and modifiers are case-insensitive. Instead of including examples here, you can query real SPF records with the above tool. If your SPF record triggers more than 10 lookups, you have to inline some of your a or mx mechanisms. Before you inline an include mechanism, make sure that you fully understand how they work. It goes without saying that you should authorize only IP addresses which belong to you or which send emails on your behalf.

▼ SPF modifiers

RFC 7208 defines two modifiers, which affect the evaluation of SPF records outside the matching behavior of mechanisms:

- redirect tells the SPF verifier to continue the evaluation with the SPF record of a different domain if none of the mechanisms in the current SPF record matched the sender's IP address. The target domain is provided after an equals sign, such as `redirect=_spf.example.com`. Unlike the include mechanism, which matches if the included SPF record results in a pass, the redirect modifier adopts the result of the redirected SPF record for the original record.
- exp allows the domain owner to specify an explanation for a fail result. If you include `exp=_exp.example.com` in your SPF record, the SPF verifier looks for a TXT record at `_exp.example.com` and returns its data to the sender in the case of a fail result. This DNS query doesn't count towards the lookup limit of 10.

Modifiers should appear at the end of an SPF record but can appear anywhere. Unknown modifiers have to be ignored so that SPF can be extended in the future. You can use any domain name in these modifiers, `_spf` and `_exp` have no special meaning.

▼ SPF macros

Instead of using a domain name after the a, mx, include, exists, and ptr mechanisms or the redirect and exp modifiers, you can use an SPF macro. SPF macros are domains, where expressions of the form `%{...}` are replaced with information from the SMTP session. Since the above tool doesn't have this information, it cannot evaluate macros. If you want to use SPF macros, you have to read the RFC. Let me just give you one example: `i` stands for the sender's IP address and `r` reverses the value, splitting on dots by default. If you are worried about breaking email forwarding when introducing SPF, you could specify a neutral result for trusted mail servers by including `?exists:%{ir}.whitelist.example.org` in your SPF record. Other use cases of macros are to provide a different policy for each user or to rate-limit messages coming from certain IP addresses. Macros can cause problems with internationalized email addresses as discussed in RFC 8616 and aggravate privacy implications by leaking sender addresses into the DNS. Such a tracking example can be found at altavista.net.

▼ HELO identity

In order to prevent mail loops, no MAIL FROM address is provided in automatic responses. In such circumstances, the address `postmaster@` followed by the domain from the HELO/EHLO command is used for SPF evaluation. The HELO identity can also be verified separately by evaluating the SPF record of the HELO/EHLO domain. Email service providers would have to configure SPF records for each of their outgoing mail servers. As far as I can tell, this is rarely done in practice. I found SPF records only for the outgoing mail servers of Outlook.com. Unless you run your mail servers yourself, this aspect of SPF is nothing to worry about.

▼ TXT size limits

DNS queries and replies use the [User Datagram Protocol \(UDP\)](#), when possible and fall back on using the [Transmission Control Protocol \(TCP\)](#) on the [transport layer](#) if necessary. Since UDP is a [connectionless communication protocol](#), no additional messages have to be sent back and forth to set up the connection, which makes it more efficient than [TCP with its handshake](#). Historically, UDP messages have been [limited to 512 bytes](#). This limit was raised in 1999 with the introduction of the [Extension Mechanisms for DNS \(EDNS\)](#). EDNS allows the sender to indicate a higher UDP payload size in a so-called [pseudo resource record](#) of the type OPT. The command-line tool `dig` displays this OPT record as follows:

```
$ dig ef1p.com +dnssec
[...]
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
[...]
```

How `dig` displays the OPT pseudo resource record. `dig` increases the UDP payload limit to 4096. As you can also see, `dig` sets the `do` (DNSSEC OK) flag of EDNS there to ask for [DNSSEC](#) records.

As an ordinary domain administrator, you likely don't have to worry about these size limits. If you operate a large email service or run into problems with certain providers, it can make sense to split one large SPF record into several smaller ones, though.

There is another limit in DNS: Each character string is [limited to 255 bytes](#). A [TXT resource record](#) can consist of several such strings, though. SPF verifiers [concatenate such strings](#) without inserting any spaces. If the user interface of your [domain name registrar](#) splits longer strings for you, you don't have to worry about this size limit either.

▼ SPF record type

There used to be an SPF record type for publishing SPF records but since this type never gained enough traction, it was discontinued in favor of the unstructured TXT record type. Nowadays, SPF records [have to be published in TXT records](#).

DomainKeys Identified Mail (DKIM)

[DomainKeys Identified Mail \(DKIM\)](#) is specified in [RFC 6376](#). ([DomainKeys](#) was a predecessor designed by [Yahoo](#) and the name survived the IETF standardization process.) DKIM allows a domain owner to take responsibility for a message by signing its body and selected header fields. Any mail server through which a message passes can add a DKIM signature. Unlike [S/MIME](#) and [PGP](#), which alter the body of a message, DKIM signatures are added in a new header field, which makes them unobtrusive for users whose mail clients don't support DKIM. Also unlike S/MIME and PGP, DKIM uses the [Domain Name System](#) as its [public-key infrastructure](#), which is secure only in combination with [DNSSEC](#). The owner of a domain can publish several [public keys](#) so that different servers can use different [private keys](#) for signing. The ability to publish several keys is also useful for introducing a new key before revoking an old one. Each public key is identified by a unique Selector within its Domain and is published in TXT record at `{Selector}._domainkey.{Domain}`. The selector can contain periods, which allows large organizations to split the namespace of their DKIM keys into several [administrative zones](#). Both the Domain and the Selector are included in the [DKIM-Signature header field](#) so that verifiers know how to retrieve the appropriate public key. Since the public key used to verify a signature is retrieved from the stated domain, a valid DKIM signature authenticates this domain.

The standard [doesn't specify](#) which [entities](#) add and verify DKIM signatures. Since DKIM keys are valid for the whole domain and cannot be restricted to individual users, messages are usually signed by the outgoing mail server after authenticating the user. If you are the only user of your domain and your email service provider doesn't support DKIM, you could add a DKIM signature to a message before submitting it to the outgoing mail server, but I'm not aware of any mail client which supports this. Since DKIM keys can be revoked at any time after a message has been delivered, DKIM signatures are typically verified by incoming mail servers, which record the result in the [Authentication-Results header field](#) for later use. While the mail client of the recipient could verify DKIM signatures as well, it would have to record the result before the signature expires. Since emails are usually synchronized to new mail clients via [IMAP](#), the DKIM-verifying mail client would have to replace the messages in the user's remote mailbox for archiving. As [noted earlier](#), Gmail displays the domain which signed a message. Similar functionality can be added to Thunderbird through [an add-on](#). Another reason for verifying DKIM signatures on incoming mail servers is to filter spam and phishing emails before they reach the user's mailbox. DKIM doesn't specify how to handle messages which don't have a valid signature from a suitable domain. It simply allows the recipient to use the [reputation of a domain](#) to assess a given message. How a domain owner can ask recipients to reject emails which don't pass domain authentication is the topic of the [next subsection](#).

If your email service provider supports DKIM for custom domains, it likely has an article on how to generate a DKIM key for your domain and how to publish the public key in your DNS zone. For example, [this guide](#) shows you how to set up DKIM for [Google Workspace](#). If you have to generate the signing key yourself, you find the instructions to do so [below](#). The following two tools help you generate and validate DKIM records. Instead of explaining the various configuration options here, you can simply hover your mouse over them to read a short description of their purpose in a [tooltip](#). For a longer explanation, you can consult [RFC 6376](#). The last three options are rarely used and just there for the sake of completeness. When [RSA](#) is used as the signing algorithm, the DKIM record can become quite long, which requires that the user interface of your domain name

registrar splits the TXT data into strings of at most 255 bytes for you. Unlike [SPE](#) and [DMARC](#), you don't have to configure a DKIM record for unused domains. Only if you have a wildcard CNAME record and don't trust the target domain not to spoof emails, you should configure a TXT record with a value of `v=DKIM1; p=at*._domainkey` in your DNS zone. The second tool uses [Google's DNS API](#) to query the DKIM records. If it finds a record, it loads its content into the first tool to make it easier to analyze and modify existing DKIM records.

Minimal output: ☐

Hash algorithms:

Key type:

Service types:

Public key:

Flags:

Just testing, ignore signatures
No subdomain in user identifier

v=DKIM1; p=

Domain: Selector:

▼ Non-repudiation

Unlike the sender's IP address, which can be verified only by the first incoming mail server, [ordinary digital signatures](#) can be verified by anyone. In comparison with [SPE](#), DKIM has the advantage that [email forwarding](#) due to [alias addresses](#) no longer breaks domain authentication. Unless a message has been modified by the relaying mail server, the final recipient can still verify the authenticity of the message. Another consequence of using digital signatures is that senders can [no longer repudiate](#) their messages. If a dispute arises from an oral conversation, it's one person's word against another person's word. For modern email, this is no longer the case and many people aren't aware of this. I don't know whether DKIM signatures would make much of a difference if an email dispute comes before a court. In the world of politics, however, the ability to cast or eliminate doubt can make a big difference. For example, when [WikiLeaks released personal emails of John Podesta](#), the chairman of [Hillary Clinton's 2016 presidential campaign](#), DKIM signatures proved the [authenticity of emails](#) that leading Democrats claimed to be fabricated by Russian intelligence agencies. On the other hand, signing all outgoing emails makes it more difficult for others to frame you for things that you've never written. While nothing speaks against deploying [SPE](#) and [DMARC](#), you should think twice about introducing DKIM and consult your legal department before doing so. If you use a large email service such as Gmail, Outlook.com, or Yahoo Mail, the provider has made the decision for you. Since they can be forced to reveal evidence anyway, you shouldn't be using them in the first place if you care about [plausible deniability](#). Instead of hosting your emails yourself, you should rather use an [off-the-record messaging protocol](#) like [Signal](#), which implements [deniable authentication](#).

▼ DKIM-Signature header field

Each DKIM signature is added in a [DKIM-Signature header field](#), which consists of `name=value` pairs separated by a semicolon. Let's look at an example before we discuss the various tags and their values:

```
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed;  
d=gmail.com; s=20161025;  
h=mime-version:from:date:message-id:subject:to;  
bh=HM9brgzIS3y+e9+sDmAHassF4IPPotkzbGUPemyxbZY=;  
b=Lx1FORL1tK47ZvPK+cDr8BRAUp+fZe1RZaxlQ4hp4qfk7jswXVcSxvntyD3VeclxiK  
XXV9Bb1WoFkyth3u0LTavPTR2w0Hb3m2IrubMIJsTsttzaV9XNsECLi2kG0Si0Ssj19  
+ZHC+Ne7+piXyzhg0GiWYDqqSfN9jmQeKKJuZLTxu0sL/UFNKzUfNdPABpcg8dLZ0C1R  
s/4u++5Zbj8T4fjp1kma+X9q+fKv5oWuYI4BbhQ6ie8g58XRidRowLZTiydocfoRC93x  
UT00JSZr4RA0EyDa5ViJTUwdzkNA6AlokRJ6JYAHoAIXtTSIFnymXjVZcBMUTMY0HoZu  
6S0A==
```

An example DKIM-Signature header field from a message sent with Gmail.

Tag	Necessity	Description
v	required	The version of the DKIM specification. The current version is 1.
a	required	The signature algorithm. The value is <code>rsa-sha256</code> or <code>ed25519-sha256</code> as introduced in RFC 8463 .
c	optional	The canonicalization of the header fields and the body separated by a slash. Either <code>simple</code> or <code>relaxed</code> .
d	required	The domain of the signer.
s	required	The selector, which identifies the used key.
h	required	A colon-separated list of the names of the header fields which are covered by the signature.

Tag	Necessity	Description
bh	required	The Base64-encoded hash of the canonicalized message body (see below).
b	required	The Base64-encoded signature. We'll discuss below how the signed hash is computed.
l	optional	How many bytes of the canonicalized message body are hashed (see below).
i	optional	The email address for which the signer takes responsibility (see below).
t	optional	The <u>Unix time</u> of when the signature was created.
x	optional	The <u>Unix time</u> of when the signature expires.

The various tags of the DKIM-Signature header field. The last four tags are less common and the [IANA registry](#) lists even more tags.

There's one tag we need to have a closer look at: h. Since the mail servers through which an email passes add additional header fields for ~~message tracing~~, ~~mail loop prevention~~, and ~~spam assessment~~, DKIM signatures cannot cover all header fields as they would be invalidated immediately otherwise. DKIM lets the signer decide which header fields it wants to sign. The signer then lists the names of all the header fields it wants to sign in the h tag in the order in which they are to be fed into the cryptographic hash function. The From header field has to be in the list and the DKIM-Signature header field which is being generated may not be included as it's fed into the hash function with an empty b tag implicitly. The header field names in the h tag are case-insensitive and the same name may appear in the list several times. The first occurrence of a header field name refers to the last header field with the same name in the message, the second occurrence to the second last header field, and so on. The h tag can list header fields which don't appear in the message. If this is the case, nothing is fed into the hash function, which means that no such header field can be added without invalidating the DKIM signature. Since the hash of the message body is included in the bh tag of the DKIM-Signature header field, it no longer needs to be fed into the hash function separately, which is clarified in [this errata](#).

Since DKIM signatures guarantee the authenticity of a message, all header fields which affect how the message is displayed to the recipient should be signed. I have no idea why Gmail signs the ~~MIME-Version header field~~ but not the ~~Content-Type header field~~. The latter is much more important as it includes the ~~multipart type~~ as well as the ~~boundary delimiter~~. Moreover, Gmail includes only the ~~Cc header field~~ in the h tag if the message has Cc recipients. By not always including Cc (and Bcc) in the list of signed header fields, such a field can be added by a relaying mail server without invalidating the signature. Header fields like ~~Sender~~ and ~~Reply-To~~ should also always be included in order to prevent others from adding them. Outlook.com and Yahoo Mail are almost as bad as Gmail in this regard. Unlike Gmail, Outlook.com's DKIM signature also covers the Content-Type, whereas Yahoo Mail includes the inexistent Reply-To header field. On a more positive note, Mailchimp signs the ~~List-Unsubscribe header field~~ as recommended by [RFC 6376](#).

Many header fields may appear at most once in valid messages. As long as DKIM verifiers ignore DKIM signatures on invalid messages, header fields like From and To don't need to be included twice in the h tag.

▼ Body and header canonicalization

Besides adding additional header fields, some mail servers also modify emails in other ways. If the verifier doesn't feed exactly the same content into the hash function as the signer, the signature is seen as invalid. Since DKIM signatures can break due to no fault of the signer, messages with only invalid signatures should not be treated differently from messages with no signatures at all. To make DKIM signatures more robust, [RFC 6376](#) defines four algorithms to canonicalize the inputs to the hash function:

- simple body canonicalization: Make sure that the body ends with a single {CR}{LF}. If the body ends with several {CR}{LF}, convert them to one. If there is no {CR}{LF} at the end of the body, insert one.
- relaxed body canonicalization: Delete spaces and tabs before {CR}{LF}, reduce all sequences of whitespace within a line to a single space, and apply the simple body canonicalization with the exception that an empty body remains empty.
- simple header canonicalization: Don't change header fields in any way.
- relaxed header canonicalization: Convert header field names to lowercase, ~~unfold~~ folded lines, remove any whitespace characters around the colon which separates the name of the header field from its value, and apply the relaxed body canonicalization to each header field.

The signer can decide how to canonicalize the header fields and the body. The format of the c tag is {HeaderCanonicalization}/{BodyCanonicalization} and its default value is simple/simple.

▼ Allow others to extend the body

The l tag limits how many bytes of the canonicalized body are included in the body hash, which is stored in the bh tag of the ~~DKIM-Signature header field~~. The idea is that the signer can allow others to extend the current message body so that ~~mailing lists~~ which add an unsubscribe footer don't invalidate the signature. Unfortunately, there are three problems with this idea:

- **Applicability:** DKIM is not MIME-aware. Anything you add to a multipart message after the last boundary delimiter is part of the epilogue and won't be shown to the user. Additionally, many mailing lists also modify the Subject line.
- **Usability:** It's not clear how partial authenticity can be conveyed to the user without causing confusion.
- **Security:** If the `l` tag is not used with utmost care, an attacker might be able to alter the original message in unexpected ways. For example, if the `Content-Type` header field is not included in the `h` tag, an attacker can move the original message into the preamble by changing the boundary delimiter. If the content is a simple HTML message, mail clients have to be strict in how they parse the message to prevent content overflows.

For these reasons, DKIM verifiers may ignore signatures which use the `l` tag.

▼ Signing messages of subdomains

Signers can specify in the `i` tag of the `DKIM-Signature` header field the identity of the user on whose behalf they sign. The `i` tag has the same format as an email address. The domain part of the `i` tag has to be the same domain as specified in the `d` tag or a subdomain thereof. Unlike the domain in the `d` tag, the domain part of the `i` tag doesn't have to exist in the DNS and the local part of the `i` tag doesn't have to be associated with any mailbox. This allows parent domains to sign on behalf of subdomains without having to publish the DKIM keys also at the subdomains. However, there's no reason to use the `i` tag for emails since DMARC ignores the `i` tag.

▼ Replay attacks for spamming

By design, DKIM protects only the body and the selected header fields of a message but not its envelope. If DKIM also covered the `RCPT-TO` addresses of the envelope, mail servers could no longer forward emails without breaking domain authentication. One of the reasons for using outgoing mail servers is to allow email service providers to limit the rate at which their users can send emails. The problem with DKIM is that any user can get a valid DKIM on a message of their choosing by sending an email to themselves. Once they have received the email, they can relay the signed message to an arbitrary number of recipients. The recipients might not see their email address in the `To` or `Cc` field but this is usually also the case for `Bcc` recipients. The email service provider who signed the message can stop such a replay attack only by revoking the corresponding signing key. If the email service provider doesn't use a different key for each user, which would increase the load on their domain name servers considerably, revoking the key immediately prevents delayed emails of other users from being delivered as well. Unless a key is compromised, it should be revoked only after around one week of no longer being in use. Even if the email service provider did use a different key for each user, it would first have to learn that a user abuses their reputation for spamming. A lot of damage might already have been done before the key is revoked and the change is propagated through the DNS. The best that public email service providers can do to counter this attack is to reject outgoing messages with a high spam score.

▼ Generating the signing key

You can generate a DKIM signing key yourself with the following commands. As far as I can tell, you can generate an ED25519 key only with OpenSSL but not with LibreSSL. I covered how to install OpenSSL on macOS in an earlier box.

OpenSSL:



```
# Generate RSA key (OpenSSL and LibreSSL):
$ openssl genrsa -out private.pem 2048
$ openssl rsa -in private.pem -pubout -out public.pem

# Generate shorter ED25519 key (only OpenSSL):
$ openssl genpkey -algorithm ED25519 -out private.pem
$ openssl pkey -in private.pem -pubout -out public.pem

# Output the public key:
$ cat public.pem
-----BEGIN PUBLIC KEY-----
[Base64-encoded public key]
-----END PUBLIC KEY-----
```

Copy the part between `-----BEGIN PUBLIC KEY-----` and `-----END PUBLIC KEY-----` into your DKIM record.

▼ Authorized Third-Party Signatures (ATPS)

Many people and companies use a custom domain without running their mail servers themselves. Instead, they delegate the delivery and the receipt of messages to email service providers. [SPF](#) makes such a delegation very easy with the [include mechanism](#), which allows email service providers to change their outgoing mail servers without involving their customers. If you want to achieve [the same with DKIM](#), you have to delegate a [DNS zone](#) in the `_domainkey` subdomain to your email service provider. The next best thing is to configure a [CNAME record](#) in this subdomain which points to the DKIM record of the email service provider. [Once again](#), the standard doesn't mention whether CNAME records can be used but this seems to be [a common practice](#). With this approach, the email service provider cannot change the selector of their signing key without involving their customers but if you set up several CNAME records, the email service provider can alternate between them.

[RFC 6541](#) introduces a simpler solution, which is called Authorized Third-Party Signatures (ATPS). It adds an additional tag with the name `atps` to the `DKIM-Signature header field`, which can be used to indicate a domain on whose behalf the message is signed. The delegator confirms the delegation with a TXT record at `{DelegateeDomain}._atps.{DelegatorDomain}`, which has to have a value of `v=ATPS1`. For example, if the domain of your organization is `example.org` and you use the email service of `example.com`, your email service provider adds a DKIM-Signature header field with `d=example.com` and `atps=example.org`. The verifier then checks whether there is a valid ATPS record at `example.com._atps.example.org`. So far, so good.

What makes ATPS unnecessarily complicated is that the `DelegateeDomain` can be hashed. The [argument for this](#) is to force the subdomain to a fixed length so that arbitrarily long third-party domain names can be prepended to the `DelegatorDomain` while remaining below [255 characters](#). Yet another tag with the name `atpsh` is added to the DKIM-Signature header field to indicate the used hash function. In the previous example, which doesn't use hashing, the tag is `atpsh=none`. If the `DelegateeDomain` is hashed, which is indicated with `atps=sha256`, the hash is encoded with [Base32](#) according to [RFC 4648](#). Since Base32 uses only the capital letters from A to Z and the numbers from 2 to 7, the encoding is better suited than [Base64](#) for case-insensitive domain names. You find an example with hashing in [Appendix A of RFC 6541](#). The RFC itself is labeled as [experimental](#) and I have no idea whether anyone actually uses ATPS.

▼ Author Domain Signing Practices (ADSP)

I said [earlier](#) that the recipient cannot know whether the sender uses DKIM when they receive a message without a DKIM signature. At least historically, a domain owner could indicate that all emails are signed with the now deprecated [Author Domain Signing Practices \(ADSP\)](#) as specified in [RFC 5617](#) by configuring a TXT record with a content of `dkim=all` at the `_adsp._domainkey` subdomain. Since ADSP is no longer in use, you don't have to configure such a record at your domains.

Domain-based Message Authentication, Reporting and Conformance (DMARC)

Increasing the security of a decentralized system is always difficult because enforcing new requirements prematurely disrupts the [reliability](#) of the system. In order to minimize the disruption for users, all changes to the system have to be [backward compatible](#). As we will see in the [next section](#), some improvements involve only two parties. Email authentication, on the other hand, involves many parties and is, therefore, quite difficult to deploy. To authenticate emails, the outgoing mail server of the sender has to implement [SPF](#) and/or [DKIM](#), email forwarders may not break the authentication, and the incoming mail server of the recipient has to verify the authentication. It only makes sense to authenticate emails if unauthentic emails are somehow penalized. In the short term, this could mean that unauthentic emails are quarantined as potential spam. In the long term, the goal should be to reject or discard all unauthentic emails even if they are delivered by a [reputable mail server](#). While it is always up to the mail system of the recipient to decide what to do with incoming messages, it can enforce domain authentication only if just a small percentage of legitimate mail is affected by it. [Domain-based Message Authentication, Reporting and Conformance \(DMARC\)](#), which is specified in [RFC 7489](#), allows domain owners to deploy [domain authentication](#) gradually, to monitor its effect on the delivery of their emails, to detect overlooked sources of legitimate mail, and to ask for strict enforcement when the amount of disruption seems acceptable.

There are three aspects to understanding DMARC:

- **Authentication:** A message is considered to be authentic if the domain of the `From` address matches the SPF-authenticated domain of the MAIL `FROM` address or the domain of a valid DKIM signature. The owner of the sending domain can require the matching to be `strict`, in which case the domains have to be identical, or `relaxed`, in which case only the [organizational domains](#) after removing any subdomains have to be the same. This is known as [identifier alignment](#) and the alignment can be configured separately for SPF and DKIM. If the `From` field consists of several addresses, which is valid according to [RFC 5322](#), the recipient can either reject the message or authenticate all domains and [apply the most strict policy](#) among the unauthentic domains.
- **Reports:** Domain owners can ask receiving mail servers to send them [aggregate reports](#) in regular intervals and [failure reports](#) for messages which failed authentication. These DMARC reports allow domain owners to monitor their deployment of domain authentication, to detect unauthorized sources of legitimate messages, such as webshops and [continuous integration systems](#), and to be informed immediately when their domain is abused for phishing.
- **Policies:** Domain owners can specify how receiving mail servers shall handle unauthentic messages. If your domain doesn't yet have a DMARC record (see below), you should start with aggregate reports and a domain policy of `none`. This allows you to be informed about authentication failures without affecting how unauthentic messages are handled. Once you're confident that you've authorized all legitimate sources of email with SPF, you should set the domain policy to `quarantine`, which requests receiving mail servers to treat unauthentic emails as suspicious.

Since it's not under your control whether your recipients use [alias addresses](#), you should move to the reject policy only once you've also deployed DKIM. Otherwise, your messages might not even reach the spam folder of your recipients if they use [email forwarding](#).

Domain owners publish their preferences in a TXT record at `_dmarc.{Domain}`. The following two tools help you generate and validate the DMARC record for your domain. Given the remarks above, most parameters should be self-explanatory. If this is not the case, you can hover your mouse over them to read a short description and all options are also documented in [RFC 7489](#), of course. Domains which aren't used to send emails from should have a DMARC record of `v=DMARC1; p=reject`. If you want to be informed about spoofing attempts, you can also include a reporting address. The second tool uses [Google's DNS API](#) to query the DMARC record of the given domain. If it finds a record, it loads its content into the first tool to make it easier to analyze and modify existing records. Even if you use the first tool to generate your DMARC record, you should check your record with the second tool as there are still many mistakes that you can make. For example, if reports shall be sent to a different domain, the report receiver has to [approve this](#). Another example is that the [subdomain policy](#) has an effect only in DMARC records of [organizational domains](#). The second tool performs more than twenty such checks and warns you about potential configuration errors.

Minimal output: ☐

Domain policy:

Reject

Subdomain policy:

Inherit

Rollout percentage:

100

SPF alignment:

relaxed

DKIM alignment:

relaxed

Aggregate reports:

Email address

Report interval:

24

Failure reports:

Email address

Report format:

Authentication Failure Reporting Format (AFRF)

Report when:

All authentication mechanisms fail (incl. alignment)

↶

🗑

↷

v=DMARC1; p=reject

Domain:

gmail.com

Load

↶

🗑

↷

▼ Organizational domain

An [organizational domain](#) is a domain which is registered with a [domain name registrar](#). For example, the organizational domain of `support.example.com` is `example.com`. While [second-level domains](#) can be registered for many [top-level domains](#), some [country-code top-level domains](#) have [special second-level domains](#). An example of this is the [.uk domain](#), which has special second-level domains, such as `.co.uk` and `.org.uk`. Unfortunately, DNS doesn't provide a mechanism to determine which domains were registered by a different organization. In particular, [SOA records](#) are [not always used at administrative boundaries](#). Therefore, the organizational domain has to be determined by consulting a list of all known public suffixes and keeping one label more than the longest suffix found in the list. The most popular [public suffix list \(PSL\)](#) is maintained by the [Mozilla Foundation](#) and available [here](#). If different email service providers use different lists, they can come to different conclusions whether two identifiers align. In order not to add [another big dependency](#), the above tool determines the organizational domain by looking for the closest SOA record.

▼ Subdomain policy

Incoming mail servers query the `_dmarc` subdomain of the domain in the From address first. If a TXT record starting with `v=DMARC1;` is found, the message is handled according to the domain policy found in the mandatory `p` tag. If no such record is found, they have to query the `_dmarc` subdomain of the [organizational domain](#) next. If a DMARC record is found, the message is handled according to the subdomain policy found in the optional `sp` tag or according to the domain policy if no subdomain policy is specified. For example, if there is a TXT record of `v=DMARC1; p=none; sp=reject` at `_dmarc.example.com`, emails from `user@example.com` are handled according to the none policy while emails from `user@support.example.com` are handled according to the reject policy, assuming that no DMARC record exists at `_dmarc.support.example.com`. If such a record does exist, emails from `user@support.example.com` are handled according to the policy specified in the `p` tag of this record. This is why subdomain policies have an effect only when specified in the DMARC record of an organizational domain. The alternative approach of removing one subdomain after another until a DMARC record is found [was considered and rejected](#) because this would allow a malicious sender to trigger dozens of DNS requests on the incoming mail server.

▼ Unix time

[Unix time](#) is the number of seconds since 1970-01-01 at 00:00:00 [Coordinated Universal Time \(UTC\)](#) without [leap seconds](#) so that every day contains exactly 86'400 seconds. Unix time is used where a concise date format is desirable. The following tool converts between unix time and the widely used [Gregorian calendar](#).

▼ Aggregate reports

Incoming mail servers which support DMARC send aggregate reports to the addresses specified in the rua tag of the sender's DMARC record in regular intervals. If no DMARC record is found for the domain in the From address, they look for a DMARC record at the sender's organizational domain. Aggregate reports are structured with the Extensible Markup Language (XML) according to this schema. They contain information about how many emails the submitter received from which IP addresses and whether these messages passed DMARC authentication – but not whether these messages were actually delivered. Spammers can also configure DMARC records and you don't want to give them insights into the effectiveness of their campaigns. An aggregate report looks as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<feedback>
  <report_metadata>
    <org_name>google.com</org_name>
    <email>noreply-dmarc-support@google.com</email>
    <extra_contact_info>https://support.google.com/a/answer/2466580</extra_contact_info>
    <report_id>4027243601387366635</report_id>
    <date_range>
      <begin>1582588800</begin>
      <end>1582675199</end>
    </date_range>
  </report_metadata>
  <policy_published>
    <domain>ef1p.com</domain>
    <adkim>s</adkim>
    <aspf>s</aspf>
    <p>none</p>
    <sp>reject</sp>
    <pct>100</pct>
  </policy_published>
  <record>
    <row>
      <source_ip>198.2.140.132</source_ip>
      <count>1</count>
      <policy_evaluated>
        <disposition>none</disposition>
        <dkim>pass</dkim>
        <spf>fail</spf>
      </policy_evaluated>
    </row>
    <identifiers>
      <header_from>ef1p.com</header_from>
    </identifiers>
    <auth_results>
      <dkim>
        <domain>gmail.mctxapp.net</domain>
        <result>pass</result>
        <selector>k1</selector>
      </dkim>
      <dkim>
        <domain>ef1p.com</domain>
        <result>pass</result>
        <selector>k1</selector>
      </dkim>
      <spf>
        <domain>mail12.mcsignup.com</domain>
        <result>pass</result>
      </spf>
    </auth_results>
  </record>
</feedback>
```

An aggregate report, which was triggered by a Mailchimp signup and sent by Google. As you can see, the message passed both DKIM and SPF authentication. However, the MAIL FROM address didn't align with the From address, which caused SPF to fail in the DMARC evaluation. As written earlier, this failure is intentional since Mailchimp wants to handle bounce messages for you, which prevents SPF from aligning.

At the time of writing, email is the only transport mechanism for DMARC reports. Aggregate reports should be compressed with GZIP as specified in RFC 1952 and sent as an attachment with the filename {RecipientDomain}!{SenderDomain}!{BeginTime}!{EndTime}!{UniqueId}.xml[.gz] in an email with the subject Report Domain: {SenderDomain} Submitter: {RecipientDomain} Report-ID: {ReportId}. All dates are provided in Unix time.

Since having compressed XML files in your inbox isn't very useful, you want to use a service which aggregates the aggregate reports for you. I use the DMARC analyzer from Postmark for my domains. Once configured, you get a weekly email with insights into the sources which send emails on behalf of your domain and the percentage of emails which passed DMARC authentication. While the service provider learns neither the local part of email addresses nor the content of emails, it learns how many emails were sent to which domains. This is sensitive metadata, which can disclose confidential business relations.

▼ Failure reports

The owner of a domain can ask incoming mail servers to send them a failure report for each message that failed one or both authentication mechanisms. Unlike aggregate reports, which are delivered periodically, failure reports are sent right after an authentication failure to the addresses in the ruf tag of the sender's DMARC record. This allows the domain owner to detect and address delivery problems quickly. Since failure reports include either the complete unauthentic message or at least its header, the domain owner can also analyze phishing attempts with a spoofed From address. In order to avert denial-of-service attacks on the domain owner, incoming mail servers are encouraged to either aggregate similar failures over short time periods or rate limit failure reporting while discarding the remainder.

The Authentication Failure Reporting Format (AFRF), which is specified in RFC 6591, is currently the only format for DMARC failure reports. It is a feedback type of the Abuse Reporting Format (ARF), which is specified in RFC 5965. The format is sometimes also called the Messaging Abuse Reporting Format (MARF), which was the name of the IETF working group. To make failure reports easy to read for machines, they are structured as follows:

```
MIME-Version: 1.0
Content-Type: multipart/report; boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: text/plain

This is an authentication failure report.

--UniqueBoundary
Content-Type: message/feedback-report

Feedback-Type: auth-failure
Identity-Alignment: [none|spf|dkim]
Original-Mail-From: attacker@example.com
Source-IP: 192.0.2.1
[...]

--UniqueBoundary
Content-Type: [message/rfc822|text/rfc822-headers]

From: attacker@example.com
To: victim@example.org
Subject: Your credit card is about to expire
[...]

--UniqueBoundary--
```

The structure of failure reports. [A|B] means either A or B and [...] stands for more header fields. multipart/report and text/rfc822-headers are specified in RFC 6522, message/rfc822 is specified in RFC 2046, and message/feedback-report is specified in RFC 5965. IANA maintains a list of feedback report header fields. The DMARC-specific Identity-Alignment header field is defined in RFC 7489.

As far as your privacy is concerned, your email service provider has your message anyway if you submit it through the outgoing mail server. If your message failed DMARC authentication because you bypassed the outgoing mail server, you shouldn't have an expectation of privacy in the first place. DMARC's failure reporting does increase the chance that your message is screened by an IT administrator, though. If you care about privacy, you should use S/MIME or PGP for end-to-end security.

▼ Report approval

Since DMARC records are publicly visible and DMARC reports can be large and frequent, you should set up a dedicated mailbox or an [alias address](#) to receive them. In order to prevent bad actors from flooding a victim's mailbox with unwanted reports by listing his or her address in the rua or ruf tag of their DMARC record, a receiving domain has to approve each domain for which it is willing to receive DMARC reports [with a special DMARC record](#) unless they belong to the same [organizational domain](#). The content of such approval records is `v=DMARC1`; and they are published as TXT records at `{PolicyDomain}._report._dmarc.{ReceivingDomain}`. For example, if the DMARC record of `example.org` includes `rua=mailto:dmarc@example.com`, there has to be an approval record at `example.org._report._dmarc.example.com`. The [above tool](#) verifies this for you. If you run a DMARC report analyzer business and want to allow anyone to direct reports to you, you can configure a [wildcard record](#) at `*._report._dmarc.{YourDomain}`. Since the local part of the report address needs no approval, you should use a dedicated domain for this. Otherwise, anyone can fill your personal mailbox with DMARC reports. Due to this approval mechanism, report emails don't have an [unsubscribe button](#). To avoid processing fraudulent reports, all report emails must pass DMARC authentication themselves.

▼ Authentication-Results header field

The [Authentication-Results header field](#) allows incoming mail servers to record the results of various authentication methods so that [email filters](#) can use them in their rules and [mail clients](#) can display them to their users without having to verify the various authentication methods themselves. It is a [trace field](#) and should therefore be added at the top of the header, ideally above the [Received header field](#) so that downstream agents can determine more easily whether the Authentication-Results header field can be trusted. Its format is as follows:

```
Authentication-Results: {Verifier} [{Version}][;  
  {Method}={Result}[ {PropertyType}.{PropertyName}={Value}]*  
]*
```

The format of the Authentication-Results as specified in [RFC 8601](#). The curly brackets need to be replaced with actual values. The content in square brackets is optional and the asterisk indicates that the preceding content can be repeated.

The Verifier is an identifier for the entity which performed the verification and the optional Version indicates which version of the field format is in use, where its current and default value is 1. [IANA](#) maintains a long list of [authentication methods and their results](#). The Method is usually `spf`, `dkim`, or `dmarc`, and the Result is usually `pass` or `fail`. The PropertyType is usually `smtp` or `header`, depending on whether the verified property is from the [SMTP envelope](#) or the [message header](#). Example:

```
Authentication-Results: mx.google.com;  
  dkim=pass header.i=@ef1p.com header.s=gm1 header.b=AT6+9nrv;  
  spf=pass (google.com: [...]) smtp.mailfrom=kaspar@ef1p.com;  
  dmarc=pass (p=REJECT sp=REJECT dis=NONE) header.from=ef1p.com
```

The Authentication-Results header field added by Google when I send an email to Gmail. [...] is the same comment as in the [SPF-Received header field](#). Gmail doesn't quite adhere to the standard. To begin with, it adds the Authentication-Results below the Received header field. Since [SPF](#) doesn't authenticate the local part of the MAIL FROM address, it [should not](#) be included in the `smtp.mailfrom` property. Furthermore, I have no idea why Gmail includes `header.i=@ef1p.com` rather than `header.d=ef1p.com` in the [DKIM](#) result. To be fair, the RFC has [one example](#) using the `d` tag and [one example](#) using the `i` tag.

Incoming mail servers can add a separate Authentication-Results header field [for each verified authentication method](#) or combine the results of all verifications into a single header field. In order to avoid [forged header fields](#), incoming mail servers [have to remove](#) all Authentication-Results header fields which use their own identifier as the Verifier. Since mail clients cannot know whether their incoming mail server supports this standard, they should interpret this header field [only on the user's request](#). Incoming mail servers could strip all existing Authentication-Results header fields from incoming messages, but this invalidates any [DKIM signature](#) which covered one of them. Furthermore, there doesn't exist a mechanism yet which allows mail clients to query the Verifier identifier of their incoming mail server. Mail clients could send an email to their own address without ever displaying it to the user in order to determine whether their incoming mail server adds such a header field and what identifier it uses. Unfortunately, Gmail doesn't add an Authentication-Results header field to messages sent to your own mailbox. In order to make use of this header field without prompting the user, who doesn't know the answer either, you have to analyze whether most messages in the user's inbox have an Authentication-Results header field with the same identifier. Unfortunately, even this technique fails if server instances use their individual hostname [as their identifier](#).

▼ Authenticated Received Chain (ARC)

[Authenticated Received Chain \(ARC\)](#) is an [experimental](#) protocol to convey [authentication results](#) across trust boundaries. It's specified in [RFC 8617](#) and it introduces the following three header fields, which are always added [as a set](#):

- [ARC-Authentication-Results](#) is a copy of the [Authentication-Results header field](#) with the addition of an [instance tag](#) with the name `i` to indicate which three ARC header fields belong to the same set.

- ARC-Message-Signature is a DKIM signature over the potentially modified message, which may not cover ARC-related and Authentication-Results header fields. The instance tag replaces DKIM's i tag and for some reason the version tag v is not defined for ARC-Message-Signature.
- ARC-Seal is a DKIM-like signature which covers all ARC-related header fields in increasing instance order. As it doesn't cover the body of the message and has a defined canonicalization, only the DKIM tags a, b, d, s, and t as well as the instance tag i can be used. A new cv tag is used to record the chain validation status. Its value can be pass, fail, or none.

Intermediary message handlers seal their authentication results so that the incoming mail server of the recipient can consider to deliver even those messages whose DMARC authentication no longer passes. These three header fields look as follows:

```
ARC-Seal: i=1; a=rsa-sha256; t=1616059926; cv=none;
        d=google.com; s=arc-20160816;
        b=MoNLEuRiwzIJ7FoSItrs3mzkBjiRhHfrADb6gVmEVHMyH1blgnpjxHqJNygEfYdVNo
        /kMFAxLbM6FPAALqyK6VGsDJQAQpHzGzVx1UQ1URugg28cAo5Kp7gSntyJYYZ1Ni/BCp
        czD915SdxwTtJ9rg0ynFUuZXfi8aAjCcZeVXGdTubDwjgs61v1KfxVf6aWMCLUkr9k9B
        JDiTrr/gyJXD1nLNPMMzRgeveIEWgqWKE32BRSDJ42i9Nq0PAHaN3k5g3z579Li9UW1N
        0obd/OCvAXD6bEYkmWtUmIIuH4HniCmC9AG7aQ9Ewko35HWwezLP7MvjLCqSYRQHeLNa
        UeaQ==

ARC-Message-Signature: i=1; a=rsa-sha256; c=relaxed/relaxed; d=google.com; s=arc-20160816;
        h=to:date:message-id:subject:mime-version:content-transfer-encoding
        :from:dkim-signature;
        bh=gmLzBJCLmp99Kb/Rm3Sh+/9143Y1eDcNI0l8V6LhSL0=;
        b=QTDYEklgqj2/0Vt7k6r2HK9Td7TVDrnmLxSs1de0ruFpbWznIpKLv2iyFbMvo9Gc0
        qKPRZi076vn0kbnGiBp5FYOP4d9LES+yR04Nx0CIj2iMJfDCUxMicQrc//ZPM7njK
        iBjNkfxDraSuFq3zh65hLYHH0if41dzLy9cPPVHqSI6luefv8MjM09tY3/5CBbg4wzIg
        8XM3RLD7lssDrz8fUgXW/nKSQav7MKzvXmnTRa43FcGvP3Aq6GQWdVl5gt8tyZWS8f
        zFH+Rn3gGG4/iGru0HGQKvaKZdrTxx43mvFL7LSv2tA+ubNqts/sV/esGQSk2r06VIAi
        wmKQ==

ARC-Authentication-Results: i=1; mx.google.com;
        dkim=pass header.i=@ef1p.com header.s=gm1 header.b=AT6+9nrv;
        spf=pass (google.com: [...]) smtp.mailfrom=kaspar@ef1p.com;
        dmarc=pass (p=REJECT sp=REJECT dis=NONE) header.from=ef1p.com
```

The ARC header fields that Gmail added to the message from which I took the Authentication-Results header field in the previous box.

Even after reading the whole RFC and skimming through the draft on recommended usage, I still question two aspects of ARC:

- **Desirability:** When it comes to spam prevention, trust based on reputation is inevitable, but when it comes to security, you want to reduce trust as much as possible. By specifying a DMARC policy of reject, a domain owner requests that incoming mail servers deliver messages from their domain based on strict authentication rather than a historic assessment of the sender's reputation. A mail server, which accepts a message from a mailing list just because it sealed its authentication results even though it broke the original DKIM signature by changing the body or the subject, subverts the purpose of domain authentication, namely knowing with certainty that the delivered messages were not spoofed. That messages modified by mail handlers are rejected because they no longer pass DMARC authentication is a feature, not a bug.
- **Necessity:** DKIM allows any domain to assume responsibility for a message. Even without ARC, a mailing list which modifies the content of a message can add its own DKIM signature and an incoming mail server is free to deliver the message based on such a signature. If an incoming mail server trusts a mailing list to authenticate received messages correctly, it can also trust the mailing list to reject messages which fail DMARC authentication. I don't understand what the advantage is of introducing yet another authentication mechanism.

If you think that I've missed the point of ARC, which might very well be the case, please let me know.

▼ DNS queries from your command line

If you don't want to use my tools to query the DNS records, you can use the dig command from your command-line interface:

Domain: gmail.com Selector: 20161025   

```
# SPF:
$ dig gmail.com txt +short

# DKIM:
$ dig 20161025._domainkey.gmail.com txt +short

# DMARC:
```

```
$ dig _dmarc.gmail.com txt +short
```

Brand Indicators for Message Identification (BIMI)

Brand Indicators for Message Identification (BIMI) is an emerging standard, which allows mail clients to display the logo of the sending company for emails which passed DMARC authentication. It is specified in various drafts. Unlike SPE, DKIM, and DMARC, BIMI is not a domain authentication mechanism. The idea is that companies can refer to an SVG image, which needs to be certified by a certification authority, in a DNS record at their domain. By ensuring that trademarks cannot be abused by scammers, BIMI has the potential to eliminate homograph attacks and phishing. Another goal of BIMI is to increase DMARC adoption among companies which value marketing more than security.

The tool below uses Google's DNS API to query the BIMI record of the given domain. BIMI records are identified with a selector just like DKIM records so that companies can use different logos for different purposes. The default selector is default. Google, Yahoo, and Fastmail are running BIMI pilots. Companies which already have a BIMI record include cnn.com, linkedin.com, and ebay.com.

Domain: Selector:

▼ BIMI DNS record

BIMI records are published as TXT records at {Selector}._bimi.{Domain}. BIMI selectors can contain periods. A selector other than default has to be indicated with a BIMI-Selector header field. Indirections with CNAME records are allowed. If no BIMI record is found at the domain of the From address, a lookup at the organizational domain is performed. BIMI records have the same format as DKIM and DMARC records. The following three tags are currently defined:

- v: The version of the BIMI standard. This tag has to come first and the only supported value is BIMI1.
- l: The location of the brand indicator. This tag is required but its value can be empty. If a value is provided, it has to be a single HTTPS URL, which resolves to a potentially GZIP-compressed SVG image.
- a: The authority evidence location. This tag is optional and its value can be empty but receivers may require it. If a value is provided, it has to be a single HTTPS URL, which resolves to a valid verified mark certificate (VMC).

BIMI records are valid only if the domain has a DMARC policy of quarantine or reject. In the case of a quarantine policy, the rollout percentage has to be 100% (pct=100). This ensures that domain owners take care that all their messages pass DMARC authentication. Additionally, the subdomain policy of the organizational domain may not be none. Including the location of the brand indicator in a new email header field was considered but rejected because the validation of the indicator couldn't be cached at the domain level. Here is an example BIMI record:

```
v=BIMI1; l=https://example.com/logo.svg; a=https://cdn.example.net/certificate.pem
```

An imaginary BIMI record with reasonably short addresses. The files can be hosted on different domains.

▼ BIMI header fields

The BIMI draft defines the following three header fields:

- BIMI-Selector: The sender can use this header field to specify a selector other than default. The format of the header field value is v=BIMI1; s={Selector}. The sender should cover this header field with a DKIM signature and the recipient should ignore this header field otherwise.
- BIMI-Location: The incoming mail server can use this header field to record the locations of the verified brand indicator and the certificate for the mail client. The format of the header field value is identical to the BIMI record. Incoming mail servers can cache valid indicators and serve them directly to their mail clients.
- BIMI-Indicator: The incoming mail server can use this header field to record the uncompressed brand indicator encoded in Base64 for the mail client after having verified the certificate. If the indicator is neither cached and served nor included in the email header by the incoming mail server, mail clients have to fetch the indicator from the sender as remote content.

The incoming mail server has to rename or remove BIMI-Location and BIMI-Indicator header fields that were added by the sender. The incoming mail server should record the result of its BIMI evaluation in the Authentication-Results header field:

```
Authentication-Results: [...]; bimi=[pass|fail|temperror|declined|skipped|none]  
header.d={{(Organizational)Domain}} header.selector={Selector} policy.authority=[pass|fail|none]
```

How the BIMI results should be recorded. [a|b] means a or b. The values are described in the BIMI draft.

Mail clients should rely on these header fields only if they know that their incoming mail server supports BIMl and strips these header fields from incoming messages. Otherwise, a mail client is vulnerable to malicious header fields included by the sender.

▼ Verified mark certificate (VMC)

This draft specifies several requirements for the X.509 certificate which is used to bind a brand indicator to a domain name:

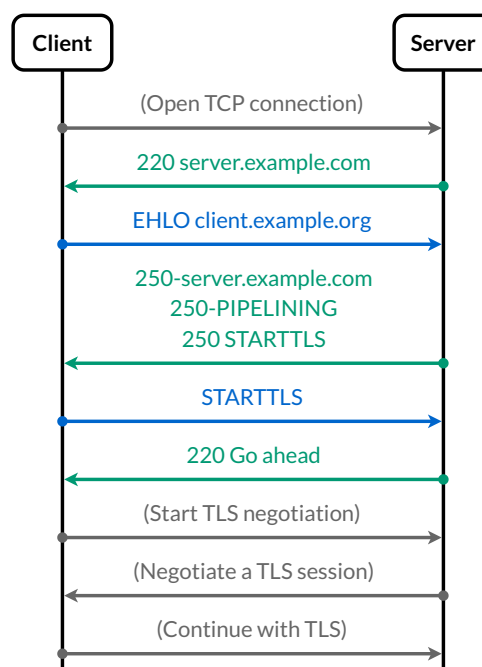
- **Certificate encoding:** The certificate must be encoded in the Privacy-Enhanced Mail (PEM) format as specified in RFC 7468. The filename specified in the a tag of the BIMl record should have a .pem extension. The file has to include the certificates of all intermediate certification authorities up to the root certification authority, whose certificate doesn't have to be included.
- **Logotype extension:** The brand indicator must be included in the logotype extension for X.509 certificates as specified in RFC 3709. The object identifier (OID) of this field is 1.3.6.1.5.5.7.1.12.
- **SVG image:** The brand indicator must be a GZIP-compressed SVG image as specified in the next box. It has to be Base64-encoded in a data URL as specified by RFC 2397 and RFC 6170.
- **Key usage:** BIMl introduces a new extended key usage with an OID of 1.3.6.1.5.5.7.3.31. This key usage must be listed in the verified mark certificate and in the certificate of the issuing certification authority.
- **Name matching:** The domain of the BIMl record with or without the {Selector}._biml. subdomain must be included in the Subject Alternative Name (SAN) field of the verified mark certificate.
- **Certificate validation:** The verified mark certificate must include a certificate revocation list (CRL) distribution point as specified in RFC 5280. The verified mark certificate must also include a signed certificate timestamp (SCT) for Certificate Transparency as specified in RFC 6962.

▼ SVG Tiny Portable/Secure profile

Scalable Vector Graphics (SVG) is a standard for, well, scalable vector graphics. Since the full standard is quite large, a subset of it has been standardized as SVG Tiny. This draft introduces a new SVG profile called SVG Tiny Portable/Secure, or SVG Tiny PS for short, for BIMl. It disallows the following features of SVG Tiny: raster images, multimedia elements, interactive elements, links to internal or external resources, scripting, and animation. The <svg> element of SVG Tiny PS documents must have the following attributes: `xmlns="http://www.w3.org/2000/svg"`, `version="1.2"`, and `baseProfile="tiny-ps"`. A <title> element must be present once as a child element of the `<svg>` element. Brand indicators should be designed to be displayed as a square or in a circle. You can use this tool to convert an existing SVG image. You find more information on the BIMl website.

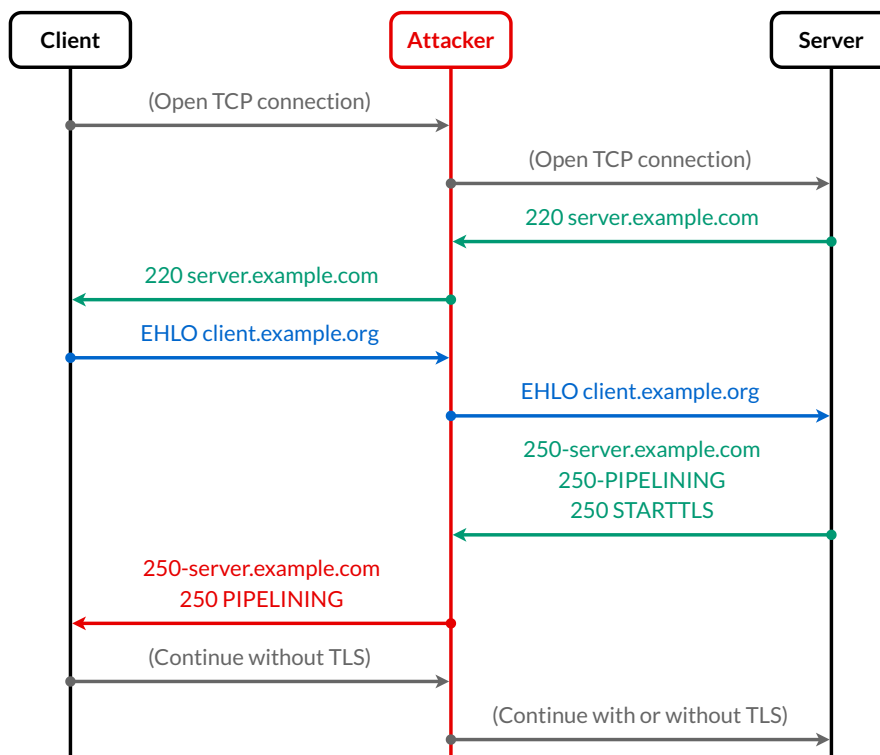
Transport security

As discussed earlier, ESMTP uses the STARTTLS extension to upgrade an insecure TCP connection to a secure TLS connection:



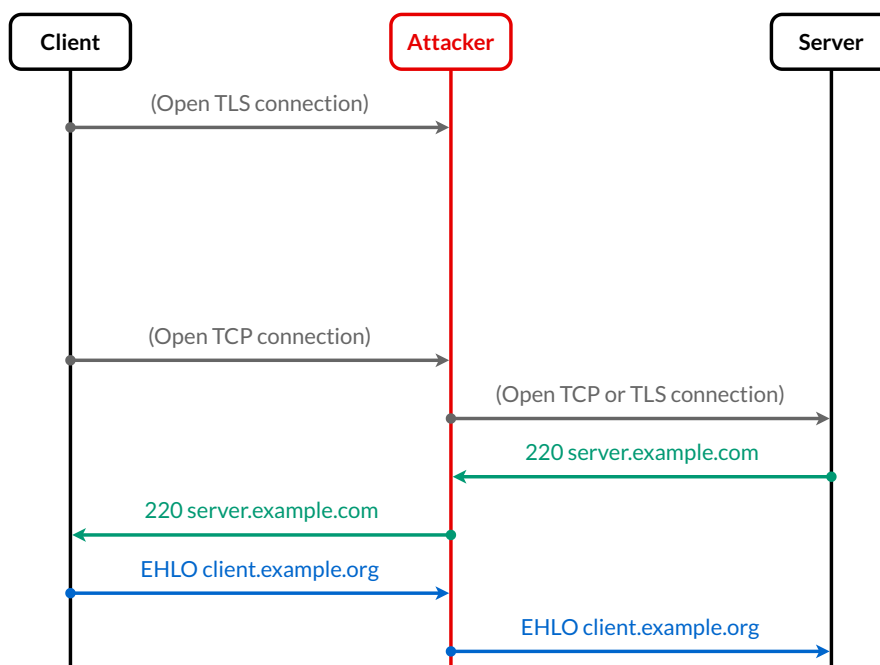
The sequence diagram of STARTTLS.

In order to remain backward compatible, the client can use the STARTTLS extension only if the server supports it. Since the server indicates support for STARTTLS over the insecure channel, an attacker who can intercept and alter the packets between the client and the server can simply strip the STARTTLS capability from the server's response to the EHLO command:



How a man in the middle can prevent the two parties from upgrading their connection to TLS.

This attack is known as STRIPTLS. The problem is not Explicit TLS but rather the opportunistic use of TLS for the sake of backward compatibility. If the client is willing to continue without the security provided by TLS, Implicit TLS suffers from the same problem:



The attacker can drop the client's TLS connection until it gives up and connects to the server with TCP.

While the opportunistic use of TLS is also a problem for submission, access, and filtering protocols, mail clients always communicate with the same few servers and should not fall back to insecure communication after the initial configuration. Additionally, cleartext is considered obsolete for email submission and access. For these reasons, we're interested only in securing ESMTP for Relay between the outgoing mail server of the sender and the incoming mail server of the recipient in this section. As discussed earlier, there are three ways to achieve secure transport in the presence of an active adversary without sacrificing backward compatibility:

1. **Previous connections:** If previous connections were secure, abort when TLS is no longer available. MTA-STS works like this.
2. **Authenticated channel:** The recipient can indicate support for TLS through an authenticated channel. DANE works like this.

3. **User configuration:** Let the user require that their messages may be delivered only with TLS. **REQUIRETLS** makes this possible.

▼ Server authentication

Without the just mentioned improvements, **ESMTP** provides only opportunistic security as defined in [RFC 7435](#). Opportunistic security provides confidentiality only towards passive attackers, who merely eavesdrop on your communication but don't interfere with it. In the presence of an active attacker, who can modify and drop network packets, opportunistic security provides neither confidentiality nor authenticity. Since opportunistic security provides no security against an active attacker anyway, many outgoing mail servers don't verify the identity of the incoming mail server they connect to but rather accept any certificate, even self-signed ones. [RFC 3207](#), which specifies the **STARTTLS** extension, explicitly allows SMTP clients to continue without server authentication because encrypting the communication for an unknown endpoint is still better than having no security at all with certainty. Requiring a valid server certificate makes sense only if the client doesn't fall back to insecure communication otherwise. Since you don't want to encrypt your communication for an active attacker, transport security extensions also have to address how clients are supposed to verify the server's identity. Given the indirection through **MX** records, they also have to define which identity is to be verified. There are two public-key infrastructures (PKIs), which have found widespread use: Vendor-configured certification authorities (CAs), using the X.509 certificate format, collectively known as PKIX, and **DNSSEC**. **MTA-STS** uses the former, **DANE** the latter.

▼ REQUIRETLS extension

As a user, you might prefer a delivery failure over an insecure delivery for certain messages. The **REQUIRETLS** extension for **ESMTP**, which is specified in [RFC 8689](#) and is not yet widely supported, allows you to require transport security between all involved mail servers when sending an email. For both submission and relay, the ESMTP server indicates its support for this extension by listing **REQUIRETLS** in its response to the **EHL0** command. If your outgoing mail server supports **REQUIRETLS**, your mail client can add **REQUIRETLS** as a parameter to the **MAIL FROM** command as follows:

```
MAIL FROM:<alice@example.org> REQUIRETLS
```

How a client asks an ESMTP server to forward the message only with TLS to other servers which support **REQUIRETLS** as well.

If a mail server accepts a message with the **REQUIRETLS** option, it may forward the message only with **REQUIRETLS** enabled. If the receiving server cannot be authenticated or doesn't support **REQUIRETLS**, the mail server has to send a bounce message with the REQUIRETLS option instead. **REQUIRETLS** may be used only if the communication is secured with TLS, the **MX** record was validated with **DNSSEC** or an **MTA-STS** policy, and the identity of the receiving server was verified with **DANE** or **PKIX**. If a message with **REQUIRETLS** enabled is forwarded by a mailing list or responded to automatically, these generated messages should also have **REQUIRETLS** enabled. Please note that **REQUIRETLS** does not provide end-to-end security: A malicious mail server has access to the content of a message and can leak it to anyone even without disabling **REQUIRETLS**.

[RFC 8689](#) also specifies the TLS-Required header field, which allows the sender to prioritize delivery over security as follows:

```
TLS-Required: No
```

How a client asks an ESMTP server to forward the message even if **DANE** or **MTA-STS** fails. At the moment, No is the only valid value.

This can be useful, for example, when you want to report a misconfigured mail server, such as an expired TLS certificate. This option is defined as a message header field rather than as an envelope parameter so that it can pass through mail servers which don't support **REQUIRETLS**. Since a mail server might implement **DANE** or **MTA-STS** without supporting **REQUIRETLS**, messages with this header field can still bounce due to a failed TLS negotiation. This header field simply allows the sender to override the TLS policy of the recipient domain. If **REQUIRETLS** is enabled for a message, the **TLS-Required** header field has to be ignored.

DNS-Based Authentication of Named Entities (DANE)


DNS-Based Authentication of Named Entities (DANE) is specified in [RFC 6698](#). [RFC 7671](#) updates and clarifies some aspects of **DANE** and [RFC 7672](#) specifies how **DANE** is applied to SMTP. **DANE** relies on **DNSSEC** for three different purposes:

- **DNS authentication:** Domain names are used to reference services, which are often provided by external service providers. Since changes are easier if the service providers can manage their address records themselves, indirections with **MX**, **SRV**, and **CNAME** records are quite common in the Domain Name System. The same is true for security-related DNS records, such as **TLSA** records, which are introduced by **DANE**. (Officially, **TLSA** is not an acronym but simply the name of the record type. Personally, I like to think of **TLSA** as Transport Layer Security Anchor.) Letting the service providers configure the necessary **TLSA** records at their domains has some advantages. However, the **TLSA** records can be trusted

only if the DNS records are authenticated with DNSSEC both in the zone of the customer and in the zone of the service provider. If the reply to the MX, SRV, or CNAME query can be spoofed by an attacker, the attacker can pose as the legitimate service provider to unsuspecting clients.

- **Downgrade resistance:** By configuring TLSA records at the appropriate subdomain, a service provider indicates that its server supports TLS. Thanks to DNSSEC's authenticated denial of existence, an attacker cannot suppress the retrieval of the TLSA records, which makes DANE resistant to downgrade attacks. Before you can deploy DANE, you have to deploy DNSSEC. If a client encounters an unsigned domain, it continues with opportunistic encryption. If a client learns from the superzone that the subzone is signed but cannot retrieve the signed TLSA records or a signed statement of their absence, it aborts the connection.
- **Trust anchor:** In order to prevent a man-in-the-middle attack, the client has to authenticate the server. Instead of relying on the traditional public-key infrastructure (PKI), DANE requires service providers to put the public key of their server or the public key of a trust anchor of their choosing into their TLSA records. DANE clients then verify whether the server's public key is confirmed directly or indirectly by one of the server's TLSA records. Relying on DNSSEC rather than on traditional certification authorities (CAs) has several advantages.

The tool below queries the MX records of the given domain and the TLSA records of each mail server. It uses Google's DNS API for the DNS queries and performs only rudimentary checks on the format of the TLSA records. It doesn't validate whether DNSSEC and DANE are deployed correctly. If you want to check this, you can use this validator. I cover how you can generate and verify TLSA records yourself below. You can deploy DANE only if your email service provider supports it. If your email service provider has configured TLSA records for their servers, all that you have to do is to enable DNSSEC on your custom domain.

Domain:   

▼ PKI comparison

DANE establishes a DNSSEC-based public-key infrastructure (PKI), which differs from the traditional PKI, also known as PKIX, in several important ways:

- **Single root authority:** While PKIX clients are shipped with numerous root certificates, all DANE clients are configured with the single public key of the root zone. The advantage of this is that all clients come to the same conclusion whether a given DANE certificate is valid, whereas a given PKIX certificate can be accepted by some clients but not by others, depending on their selection of root certificates. You don't want email to bounce just because the certificate of an incoming mail server is not accepted by your outgoing mail server. If this happens, you cannot do anything about it as a user.
- **Strict issuing constraints:** While PKIX certification authorities can be constrained in what domain names they can attest, this feature is rarely used in practice. Since any PKIX certification authority can issue a certificate for any domain, a single compromised certification authority compromises the security of the whole system. There was an attempt to address this problem for HTTP, but it caused such severe problems that it was not successful. DNSSEC, on the other hand, is strictly hierarchical: Any domain can certify only its subdomains. This gives the administrators of the root zone an awful lot of power but when it comes to eligible certification authorities for a given domain, the risk of compromise is not shared among them but rather accumulated across them. In computer security, the fewer parties you have to trust, the better. Apart from the root zone, you decide which organization you want to trust when choosing the top-level domain of your domain name.
- **Easier domain validation:** Most websites are secured with domain-validated certificates, where certification authorities verify only that you control the domain for which you request a certificate. How your control over the domain is verified is up to the certification authority but it usually involves publishing a nonce in a TXT record at your domain or at a certain path on your web server. While the non-profit organization Let's Encrypt solved the hassle of having to do this manually with the Automatic Certificate Management Environment (ACME) protocol, PKIX entails a parallel infrastructure to the Domain Name System (as long as we ignore extended-validation certificates, which also assert the legal entity of the domain owner). DANE gets rid of this additional loop by letting you publish the public key of your server instead of a nonce at your domain and letting clients do the domain validation themselves instead of relying on a certification authority to do this for them. While you no longer have to pay for PKIX certificates thanks to Let's Encrypt, you can configure TLSA records for your servers at no additional costs (assuming your domain name registrar supports DNSSEC at no additional costs).
- **Clear domain bindings:** As we will see soon, there are various ways to include domain names in X.509 certificates. While DANE supports different modes of operation and inherits the problem in some of them, it paves the way for a truly simple domain-based public-key infrastructure. Instead of having to indicate the right name to the server and to intersect the set of certified domain names with the set of acceptable domain names, DANE clients can just compare the server's public key with its TLSA records. Complexity is not just the enemy of security, it also leads to partial implementations of a standard, which in turn leads to a mess of incompatibilities.
- **Cached intermediate certificates:** TLS servers send their certificate with all intermediate certificates each time a client connects to them unless the client uses a resumed TLS handshake or the cached information extension as specified in RFC 7924. In the case of DANE, a client fetches the chain of public-key assertions to the root zone separately through the DNS, which has the advantage that they are automatically cached. One downside of this is that you have to publish new TLSA records at least one validity period before using the corresponding keys.
- **Key revocation:** Due to their complicated issuance, PKIX certificates are usually valid for months or years. If a PKIX key is compromised, you can revoke the corresponding certificate only for clients who support either Certificate Revocation Lists (CRL), as specified in RFC 5280, or the Online Certificate Status Protocol (OCSP), as specified in RFC 6960. DNSSEC signatures, on the other hand, are valid until

they expire and cannot be revoked. While you can remove a compromised DANE key immediately from your DNS zone, an attacker can replay the signed TLSA records until the RRSIG signature expires. For this reason, the validity of DNSSEC signatures should be in the order of days or at most weeks. If you cannot ensure that your zone is signed reliably in relatively short intervals, you should deploy neither DNSSEC nor DANE.

It's important to note that DANE changes only how TLS clients verify server certificates. DANE also uses X.509 certificates and no modifications are needed in TLS server software.

▼ **TLSA record type**

The TLSA record type is specified in RFC 6698. A TLSA record consists of the following four fields:

1. **Certificate usage:** This field captures two separate pieces of information in a single number. On the one hand, the number indicates whether the certificate referenced by the TLSA record belongs to a trust anchor, which certified the public key and the identity of the end entity, or to the end entity, i.e. the server to which the client wants to connect. On the other hand, the number also indicates whether a valid path to a PKIX certification authority which is trusted by the TLS client has to exist or whether the certificate is to be trusted just because it is referenced in the TLSA record. These certificate usages apply only to X.509 certificates in DER encoding.

Verification	Trust anchor	End entity
PKIX & DANE	PKIX-TA (0)	PKIX-EE (1)
DANE	DANE-TA (2)	DANE-EE (3)

The four certificate usages with the acronyms as introduced in RFC 7218.

2. **Selector:** This field specifies which part of the certificate is referenced by the TLSA record. The possible values are 0 for the full certificate and 1 for the SubjectPublicKeyInfo (SPKI), which consists of the public-key algorithm and the subject's public key. The latter does not cover the names of the issuer and the subject, the validity period or any certification constraints.
3. **Matching type:** This field specifies how the selected content is presented in the certificate association field. 0 means that the selected content is included as is in the last field, 1 means that the certificate association data is the SHA-256 hash of the selected content, and 2 means that the SHA-512 hash is used instead.
4. **Certificate association data:** This field contains the data which the certificate has to match according to the values in the selector and matching type fields.

TLSA records have a binary format. The first three fields are unsigned 8-bit integers, where 255 is reserved for private use and the remaining values are unassigned. IANA maintains a registry for these parameters. The certificate association field has to be represented as a string of hexadecimal characters. Here is an example:

```
3 1 1 76bb66711da416433ca890a5b2e5a0533c6006478f7d10a4469a947acc8399e1
```

One of the TLSA records at `_25._tcp.mail.protonmail.ch`. (I cover the location of the record in the next box.)

RFC 7671 specifies how DANE clients must verify the server's certificate depending on the certificate usage of the server's TLSA record. As you can see in the table below, clients ignore all information from the server's certificate except the SubjectPublicKeyInfo in the case of DANE-EE. In particular, the certificate can be issued for any name and the certificate is accepted even if it has expired. Since the certificate belongs to the end entity itself, it doesn't have to have a valid certificate chain. In the case of the other certificate usages, certification constraints such as maximum path length and name space restrictions have to be verified. If a server hosts the service for different customers under several domain names, it should support the TLS extension Server Name Indication (SNI) in order to return the right certificate chain to each client.

Requirements	PKIX-TA	PKIX-EE	DANE-TA	DANE-EE
Name matching	✓	✓	✓	✗
Expiration date	✓	✓	✓	✗
Certificate chain	✓	✓	✓	✗
Certification constraints	✓	✓	✓	✗
<u>Server Name Indication (SNI)</u>	✓	✓	✓	✗

Which checks clients have to perform for each certificate usage according to RFC 7671.

Given that DANE-EE uses almost none of the information in the certificate, [RFC 7250](#) specifies a TLS extension to use just the raw public key instead of the complete certificate of the server during the [handshake](#). For the same reason, it is recommended to combine a certificate usage of 3 (DANE-EE) with a selector of 1 (SPKI). On the other hand, the selector should be 0 to match the full certificate if the TLSA record references the certificate of a trust anchor with 2 (DANE-TA) so that any certification constraints are also covered by the TLSA record. The certificate usages PKIX-EE and PKIX-TA provide more security only if an [application layer protocol](#) forbids the certificate usages DANE-EE and DANE-TA. Otherwise, an attacker who compromised the DNS zone can simply change the certificate usage to DANE-EE or DANE-TA in order to [bypass the additional PKIX verification](#).

Recommendation	Certificate usage	Selector	Matching type
<u>SHOULD</u>	DANE-EE (3)	SPKI (1)	SHA-256 (1)
<u>MAY</u>	DANE-TA (2)	Full (0)	SHA-256 (1)
<u>SHOULD NOT</u>	-	-	Complete (0)
<u>SHOULD NOT</u>	PKIX-EE (1)	-	-
<u>SHOULD NOT</u>	PKIX-TA (0)	-	-

Which combinations of DANE parameters you should and should not use.

The verbs in the recommendation column are specified in [RFC 2119](#).

The last two rows are applicable only to [SMTP for Relay](#) on port 25.

The advantage of DANE-TA over DANE-EE is that certificates can be updated without having to change the TLSA records. When using DANE-TA, the server should include the certificate of the trust anchor in its chain of intermediate certificates, which is sent to connecting clients. Otherwise, clients cannot verify the server certificate unless the parameters are 2 0 0, which is not recommended.

▼ TLSA record location

In order to keep the TLSA records of different services apart, they are [stored at a subdomain](#), which includes the [port number](#) and the [transport layer protocol](#) of the service: `_{PortNumber}._{TransportProtocol}._{ProviderDomain}`. While the `TransportProtocol` is usually `tcp`, it can also be `udp` in the case of [Datagram Transport Layer Security \(DTLS\)](#) or `sctp` in the case of the [Stream Control Transmission Protocol \(SCTP\)](#). For example, the Dutch [Internet Standards Platform](#) has a TLSA record for [ESMTP](#) at `_25._tcp.internet.nl` and a TLSA record for [HTTPS](#) at `_443._tcp.internet.nl`.

The `ProviderDomain` is the hostname of the server which actually provides the service. It is determined by resolving CNAME records to the canonical domain and following service-specific records, such as MX and SRV records. Requiring the TLSA record to be located at the `ProviderDomain` has three advantages:

- **Easier certificate changes:** By having the service provider configure the TLSA record at their domain, they can change the server certificates without involving their customers.
- **No unnecessary records:** Such a delegation of authority is also possible with CNAME records but this would require domain owners to point to each of their service providers twice: once for the hostname of the server and once for the TLSA record of the server. Requiring clients to resolve DNS indirections before making the TLSA lookup avoids this duplication. In the example above, both `_25._tcp.internet.nl` and `_443._tcp.internet.nl` point to a different domain name with a CNAME record. (The TLSA record for ESMTP is at `_25._tcp.internet.nl` only because `internet.nl` points its MX record to itself.)
- **Easier virtual hosting:** The `ProviderDomain` is also used for verifying the [subject of the certificate](#). By using the domain of the provider rather than the customer, the service provider can use a single certificate for an arbitrary number of customers. Otherwise, the service provider would have to obtain a separate certificate for each customer. Since [X.509 certificates](#) are not constrained to a single service such as email, this is not just undesirable from an operational perspective but also from a security perspective in the case of PKIX-TA and PKIX-EE.

A disadvantage of this approach is that it's up to your service provider to support DANE, whereas you can deploy [MTA-STS](#) yourself if your service provider uses a certificate issued by a widely accepted certification authority. Another consequence of domain-based authentication is that DANE (and MTA-STS) cannot be used with [address literals](#), such as `user@[192.0.2.1]`.

▼ Multiple TLSA records

A server can have several TLSA records and its certificate has to be authenticated by just one of them. There are two situations in which you want to use multiple TLSA records:

1. **Key rotation:** Cryptographic keys have to be replaced from time to time. Since it takes [some time](#) for new records to propagate through the Domain Name System, the TLSA record for a new key has to be published at least one validity period before it can be used. The TLSA

record for the old key/certificate can be removed only once the new key is being used.

2. **Algorithm agility:** DANE clients don't have to support all TLSA parameters and combinations thereof. A service provider can publish several TLSA records with different parameters for the same key/certificate, which allows DANE clients to rely on the strongest TLSA record which they can use. For example, DANE clients should but don't have to support SHA-512. If SHA-256 is deprecated at some point in the future, service providers can publish the same key/certificate in TLSA records with a matching.type of 1 (SHA-256) and 2 (SHA-512). Clients which support the latter will use the latter record, all the others will continue to use the former one. RFC 7671 requires that each published combination of TLSA parameters covers the certificate chain in use so that DANE clients can abort the connection if the server cannot be authenticated with one of the usable TLSA records.

▼ Name matching

If a certificate usage other than DANE-EE is being used, DANE clients have to verify that the validated certificate matches the server's identity. In order to do so, they carry out the following, rather complicated and boring procedure:

1. **Determine the TLSA domain:** The TLSA lookup is usually performed on the server's domain after resolving any CNAME, MX, and SRV records. RFC 7672, however, makes things a bit more complicated for ESMTP. First of all, the domain found after the @ symbol in the email address is CNAME-expanded. If the whole expansion was secure, you look up its MX records. Any domain name listed in an MX record has to resolve directly to A/AAAA records according to RFC 2181 and RFC 5321. If the expanded @ domain has no MX records, you fall back to its A/AAAA records. The TLSA domain is determined as follows:

Non-expanded @ domain	Expanded @ domain	MX domain	TLSA domain
Secure CNAME record	Secure MX records	Secure A/AAAA and TLSA records	MX domain ✓
Secure CNAME record	Secure A/AAAA and TLSA records	–	Expanded @ domain ✓
Secure CNAME record, secure TLSA records	Insecure A/AAAA records	–	Non-expanded @ domain ⚠
Secure MX records	–	Secure A/AAAA and TLSA records	MX domain ✓
Insecure MX records	–	Secure A/AAAA and TLSA records	MX domain ⚠
Secure A/AAAA and TLSA records	–	–	Non-expanded @ domain ✓

In all other cases, DANE does not apply. I've marked the unexpected cases with ⚠.

2. **Determine the set of acceptable domains:** DANE clients accept a certificate issued for the TLSA domain or any domain to its left in the above table. In the first case, the set of acceptable domains consists of the MX domain, the expanded @ domain, and the non-expanded @ domain. RFC 7672 allows ESMTP clients to use DANE even if the MX records were in a zone whose records aren't authenticated with DNSSEC. In this second last case according to the above table, the MX domain is the only acceptable domain. Accepting the @ domain in all other cases allows different MX hosts to use the same certificate. Since the MX records couldn't be trusted before DNSSEC, pre-DANE ESMTP clients often look for the @ domain in server certificates. Requiring DANE-capable ESMTP clients to accept the @ domain as well helps with backward compatibility.
3. **Determine the set of certified domains:** In order to receive the right certificate chain from the server, DANE clients have to send the TLSA domain to the server with the Server Name Indication (SNI) extension. The server certificate can have several domain names in the Subject Alternative Name (SAN) field. Only if this field contains no domain names, clients may consider the subject's Common Name (CN). RFC 6125 goes into more detail about subject naming in PKIX certificates.
4. **Intersect the set of certified domains with the set of acceptable domains:** The certificate matches the server's identity if one of the acceptable domains is included in the set of certified domains. Even this step is not trivial as the certified domains may have the wildcard character * as the leftmost label. For example, *.example.com matches mail.example.com.

If you implement an ESMTP client, you can avoid all of this complexity by supporting only the certificate usage DANE-EE.

▼ Client behavior

A DNS lookup with DNSSEC enabled leads to one of the following three results as specified in RFC 4035:

- **Secure records:** The returned resource records have a valid DNSSEC signature, where the zone's signing key is confirmed recursively by the zone above up to the root zone.
- **Insecure records:** The returned resource records belong to a zone which is either not signed or signed with an algorithm which the client doesn't support. Either case has to be confirmed by a secure parent zone. If a zone is insecure, all of its subzones are also insecure.
- **Failure:** The returned resource records belong to a zone which is signed but they lack a valid DNSSEC signature. The lookup also fails if the client doesn't receive a response to one of its queries.

If a lookup involves indirections with CNAME, MX, or SRV records, all the indirections have to be secure for the expanded domain to be considered secure. Depending on the outcome of the TLSA lookup, DANE clients behave as follows:

DNS lookups	TLSA availability	TLSA usability	Client behavior
All secure	Available	Usable	DANE-authenticated TLS
All secure	Available	Unusable	MX: Unauthenticated TLS <u>⌋</u> SRV: PKIX-authenticated TLS <u>⌋</u>
All secure	Unavailable	–	Pre-DANE TLS
Any insecure	–	–	Pre-DANE TLS (or DANE-authenticated TLS if TLSA records are available and secure <u>⌋</u>)
Any failure	–	–	Continue with next host; delay or abort if last host

How the client has to handle the various situations according to [RFC 7672](#) and [RFC 7673](#).

A couple of remarks on the above table:

- “All secure” means that all the DNS lookups to determine the TLSA domain as well as the TLSA lookup are secure, including CNAME indirections of the TLSA record itself.
- “Unusable” means that the DANE client does not support the parameters of the TLSA record. By configuring TLSA records, DANE clients know that the server supports TLS and must not continue without it.
- “Unavailable” means that the client received an authenticated denial of existence of the TLSA records.
- “Pre-DANE TLS” stands for whatever clients did before the introduction of DANE. In the case of ESMTP, this means opportunistic TLS. In the case of JMAP, which is used only with TLS, this would mean PKIX-authenticated TLS once JMAP/HTTPS adopts DANE.
- In the unexpected cases of the previous box, DANE clients can enforce DANE-authentication even if some of the DNS lookups were insecure.
- MX records must be sorted by preference and SRV records must be sorted by priority and weight without considering which hosts use DNSSEC and DANE.
- If you want to test your configuration, you can deploy DANE only for your first MX host. If there are any misconfigurations, DANE clients will deliver the message to the next MX host. Once your DANE configuration works, you should deploy DANE for all your hosts. Otherwise an attacker can cause DANE clients to continue without server authentication or even TLS.
- DANE is backward compatible for all domains which deploy DNSSEC correctly. As long as a domain deploys neither DNSSEC nor DANE with TLSA records, DANE clients communicate with its servers as if DANE doesn't exist.
- In order to stop using DANE, you just have to remove the TLSA records.

▼ How to generate a TLSA record

Before you deploy DANE, make sure that all involved domains support DNSSEC. And even if you don't use DANE, you should activate the transfer lock on all your domains. You can generate the certificate association data with the following command:

Certificate file: Selector: Matching type:   

```
$ openssl x509 -in certificate.pem -noout -pubkey | openssl pkey -pubin -outform DER | openssl sha256
```

▼ How to verify a TLSA record

OpenSSL but not LibreSSL has the option `-dane_tlsa_domain` to configure the TLSA domain and the option `-dane_tlsa_rrdata` to provide one or several TLSA records. I covered how to install OpenSSL on macOS in an [earlier box](#). The [above tool](#) generates the OpenSSL command for you. Here is an example with [ProtonMail's](#) TLSA records at the time of writing:

OpenSSL:



```
$ openssl s_client -starttls smtp -connect mail.protonmail.ch:25 -verify_return_error -dane_tlsa_domain "mail.protonmail.ch" -dane_tlsa_rrdata "3 1 1 6111a5698d23c89e09c36ff833c1487edc1b0c841f87c49dae8f7a09e11e979e" -dane_tlsa_rrdata "3 1 1 76bb66711da416433ca890a5b2e5a0533c6006478f7d10a4469a947acc8399e1"
[...]
---
SSL handshake has read 5166 bytes and written 433 bytes
Verification: OK
Verified peername: *.protonmail.ch
DANE TLSA 3 1 1 ...8f7d10a4469a947acc8399e1 matched EE certificate at depth 0
---
[...]
```

Make sure that you use the option `-verify_return_error`. Otherwise, OpenSSL continues the session even if there was a verification error. If you feel adventurous, you can also generate the certificate association data for a TLSA record with the parameters 3 1 1 using the following command and compare the output with the TLSA record yourself:

```
$ echo 'QUIT' | openssl s_client -starttls smtp -connect mail.protonmail.ch:25 2> /dev/null | openssl x509 -noout -pubkey | openssl pkey -pubin -outform DER | openssl sha256
76bb66711da416433ca890a5b2e5a0533c6006478f7d10a4469a947acc8399e1
```

How to compute the certificate association data directly from the server certificate. 2> /dev/null suppresses the [error output](#).

▼ DANE on outgoing mail server

Configuring TLSA records for your incoming mail servers is only half the battle. If you run incoming mail servers, you almost certainly also operate an outgoing mail server. Having TLSA records for your incoming mail servers allows others to deliver email securely to you but you also want to make sure that your messages are delivered securely to others. Even if you don't configure TLSA records for your incoming mail servers, you should activate DANE on your outgoing mail server so that it authenticates the incoming mail servers of your recipients before delivering your messages. Both [Postfix](#) and [Exim](#) support DANE. [This guide](#) shows you how to configure them. The number of mail servers with TLSA records is [rising steadily](#). Here are [some statistics](#) for the [.nl top-level domain](#).

▼ HTTP Public-Key Pinning (HPKP)

As mentioned in the [PKI comparison](#), any PKIX certification authority can usually issue a certificate for any domain and thus a single compromised certification authority compromises the security of the whole system. This problem exists wherever PKIX is used, including the [Web](#). On the Web, it was addressed by [HTTP Public-Key Pinning \(HPKP\)](#), which is specified in [RFC 7469](#) and [is no longer supported](#) by any relevant browser. When accessing a website with HTTPS, the server could include a [Public-Key-Pins header field](#) in its response to list hashes of [public keys](#). Browsers then stored these hashes for the specified validity period, during which they required that a public key in the server's certificate chain matched one of the hashes pinned for the given domain. Each pin is the [Base64-encoded SHA-256 hash](#) of the [DER-encoded SubjectPublicKeyInfo \(SPKI\)](#). The public key can belong to the server, an intermediate certification authority, or a root certification authority. HPKP thus corresponds to DANE with PKIX-TA or PKIX-EE as the certificate usage, SPKI as the selector, and SHA-256 as the matching type with another encoding of the hash. But unlike DANE, the information is conveyed in a previous HTTPS response instead of the DNS, which means that connections with expired or missing pins are not protected by HPKP. The standard requires that one of the pins matches no certificate in the certificate chain in order to force administrators to include a [backup pin](#). This did not prevent people from making their domains unusable by losing access to their keys, which could also happen due to [ransomware](#). Additionally, an attacker who could compromise the first secure connection could make the legitimate website inaccessible. While HPKP was nice in theory, it caused so many problems in practice that it was discontinued. Let's look at an example:

```
Public-Key-Pins: max-age=2592000; includeSubDomains;  
pin-sha256="E9CZ9INdbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";  
pin-sha256="LPJNul+wow4m6DsxbninhswHlwfp0JecwQzYp0LmCQ=";  
report-uri="https://other.example.com/pkp-report"
```

A Public-Key-Pins response header field with a validity period of 30 days and a [report URI](#) for [validation failures](#).

Mail Transfer Agent Strict Transport Security (MTA-STS)

[Mail Transfer Agent Strict Transport Security \(MTA-STS\)](#) is specified in [RFC 8461](#). MTA-STS is a [PKIX](#)-based [alternative to DANE](#) for those who cannot or don't want to deploy [DNSSEC](#) on their domain. It lets receiving domains indicate their support for PKIX-authenticated TLS with the following two resources:

- **DNS record:** A TXT record is used to inform the sender that the receiving domain has an MTA-STS policy and whether the policy has been changed since the last time the sender retrieved it. Since small DNS records are retrieved with [UDP](#), this is much faster than retrieving the policy file, which requires a [TCP](#) and a [TLS handshake](#).
- **Policy file:** The sender fetches the MTA-STS policy with [HTTPS](#) from the receiving domain. The MTA-STS policy indicates what the sender shall do if it cannot authenticate the incoming mail server of the recipient with the presented PKIX certificate. Since MTA-STS doesn't require that DNS records are authenticated with DNSSEC, the policy file is also used to authenticate the MX records of the receiving domain. This allows clients to match the presented certificate against [the name of the mail server](#).

The tool below queries the MTA-STS record and the policy file of the given domain. It uses [Google's DNS API](#) for the DNS query and the [email tracking server](#), which I've deployed on [Heroku](#), as a [proxy server](#). This is necessary because the policy file is usually served without the header field which is required for [cross-origin resource sharing \(CORS\)](#). As you can see in [its source code](#), my proxy server doesn't store anything but [Heroku logs the last 1'500 requests](#), which includes the queried domain and your IP address. I don't persist the [log file](#) but I might check it from time to time for troubleshooting. The tool checks the syntax of the DNS record and the policy file but it verifies neither the MX records nor whether the mail server has a valid PKIX certificate.

Domain:

Query



▼ Comparison to DANE

[MTA-STS](#) differs from [DANE](#) in several important ways:

- **PKIX instead of DNSSEC:** MTA-STS relies on the traditional [public-key infrastructure \(PKI\)](#) based on [X.509 certificates](#) issued by pre-configured [certification authorities \(CAs\)](#). DANE, on the other hand, relies on [DNSSEC](#) to bind [public keys](#) to domains. I've compared the two approaches in a [previous box](#).
- **@ domain instead of MX domain:** The [TXT record](#) and the [policy file](#) of MTA-STS are stored on the domain after the @ symbol in the recipient's email address, whereas the [TLSA records](#) of DANE are configured on the domains listed in the MX records (at least in [ordinary setups](#)).
- **No strict downgrade resistance:** Unlike DANE, which provides downgrade resistance thanks to DNSSEC's authenticated denial of existence, the DNS record and policy file of MTA-STS can be [censored by an active attacker](#). MTA-STS provides downgrade resistance only for the duration for which the recipient's policy is cached. As we will see [later](#), MTA-STS has a provision which allows administrators to detect policy refresh failures.
- **Different test modes:** If you want to test your deployment of DANE, you can configure TLSA records for just some of your mail servers. MTA-STS, on the other hand, has an explicit mode for testing, which is intended to be used in combination with [SMTP TLS Reporting \(TLSRPT\)](#). Such a test mode is necessary because the MTA-STS policy affects the whole domain.
- **No support from email service providers required:** If your incoming mail server uses a valid PKIX certificate for its domain, you can deploy MTA-STS on your custom domain without explicit support from your email service provider. It's unlikely that your email service provider stops using valid PKIX certificates for their servers unless they start using DANE. In order to be on the safe side, you may want to contact your email service provider in any case.

▼ Coexistence with DANE

As long as some [ESMTP](#) clients support [DANE](#) but not [MTA-STS](#) and vice versa, you can deploy both DANE and MTA-STS on your domain to achieve the best security. [RFC 8461](#) states explicitly that MTA-STS may not override a failed DANE validation. Unfortunately, it doesn't say anything about whether DANE can override a failed MTA-STS validation:

DANE	MTA-STS	Result
✓	✓	✓
✗	✓	✗
✓	✗	?
✗	✗	✗

Can DANE override MTA-STS validation?
Unfortunately, the standard is silent on this.

In my opinion, a successful DANE authentication should take precedence over a failed MTA-STS validation as this would allow domain owners to deploy transport security without the support of their email service provider by deploying DNSSEC and MTA-STS. If the email service provider switches to DANE with self-signed certificates, clients which support DANE and MTA-STS would continue to deliver emails to this domain even if MTA-STS validation now fails.

If you deploy either DANE or MTA-STS, I advise you to deploy DANE, assuming that your domain name registrar or name server provider supports DNSSEC. DANE is more elegant and provides better security than MTA-STS. The only reason we have DANE is because some large companies are reluctant to deploy DNSSEC. All I know is that Google wants to get rid of 1024-bit RSA and focuses its efforts on Certificate Transparency. But given that ECDSA, Ed25519, and Ed448 can be used for DNSSEC and that both the root zone and the .com domain use 2048-bit RSA at least for their key-signing keys, weak cryptography is no longer really an argument against DNSSEC, especially given that DNS records aren't authenticated at all otherwise.

▼ MTA-STS DNS record

When delivering an email, an outgoing mail server which supports MTA-STS checks whether it has a non-expired policy for the domain after the @ symbol in the recipient's email address. If this is not the case, it queries for a TXT record at the `_mta-sts` subdomain of this domain. An MTA-STS record is a semicolon-separated list of key-value pairs encoded in ASCII, where the keys and values are separated by an equals sign. MTA-STS records have two required fields:

- `v`: The version of the MTA-STS standard. This field has to come first and the only supported value is STSv1 at the moment.
- `id`: A unique identifier for the domain's current MTA-STS policy. It consists of at least 1 and at most 32 letters and digits.

At most one TXT record may be returned per version. Indirections with CNAME records are allowed. If a valid MTA-STS record exists but the policy file cannot be loaded, the outgoing mail server has to deliver the message as if the recipient's domain has not implemented MTA-STS, unless the outgoing mail server still has a valid policy in its cache, in which case it has to apply this policy. If you configure MTA-STS for your domain, you should always update the policy file before updating the TXT record so that senders don't cache the old policy by mistake. If you cannot use the above tool, here is an example record:

```
v=STSv1; id=20190429T010101;
```

The TXT record at `_mta-sts.gmail.com` in April 2021.

▼ MTA-STS policy file

If the MTA-STS record contains an unknown `id` value for the given domain, the outgoing mail server retrieves the new MTA-STS policy from `https://mta-sts.{Domain}/.well-known/mta-sts.txt`, where `Domain` is the domain after the @ symbol in the recipient's email address. The `Domain` is taken as is without proceeding with the parent domain if the queried resource is not found for both the DNS and the HTTPS lookup. As specified in RFC 8615, IANA maintains a registry for the `.well-known` directory in order to prevent name collisions among unrelated standards. A subdomain is used to allow CNAME indirections. Moreover, the required DNS record prevents sites which allow untrusted users to claim subdomains from falling victim to denial-of-service attacks with malicious policies. Examples are <https://mta-sts.github.io> and <https://mta-sts.blogspot.com>.

The policy file is fetched with a GET request and the response must have a status code of 200 and a content type of `text/plain`. Its content is parsed as ASCII, which means that internationalized domain names have to be Punycode-encoded. An MTA-STS policy is specified with one key-value pair per line, where the keys and values are separated by a colon and the lines are separated by `{CR}{LF}` or just `{LF}`. Each policy has to contain the following four fields, which may appear in any order:

- `version`: The version of the MTA-STS standard. Its value has to be STSv1.
- `mode`: What the ESMTP client shall do if it cannot authenticate the incoming mail server. The possible values are:

- **enforce**: The client must abort the connection if it cannot negotiate TLS with a valid server certificate. It continues with the next server which matches one of the mx keys and delays the delivery of the message when all of them fail validation. Before failing permanently, the client has to check via DNS and then HTTPS for an updated policy.
- **testing**: MTA-STS validation failures don't affect the delivery of messages but they should be reported if both the sender and the recipient support SMTP-TLS Reporting (TLSRPT).
- **none**: The receiving domain doesn't have an MTA-STS policy. In order to detect downgrade attacks, MTA-STS clients should refresh cached policies when they are still valid. If clients cannot refresh a policy while the previous policy is still valid, they should alert the administrator unless the mode of the cached policy is none. For this reason, you should publish an MTA-STS policy with a mode of none until all previous policies have expired before removing MTA-STS on a domain.
- **max_age**: For how many seconds this policy can be cached. In order to give attackers fewer opportunities for downgrade attacks, this value should be as high as is practical. The maximum value is 31'557'600, which corresponds to 365.25 days.
- **mx**: This field authenticates the incoming mail servers of the receiving domain. Unlike the MX records of the domain, which are not authenticated when DNSSEC isn't used, the policy is fetched via HTTPS from a server with a valid PKIX certificate. This field may occur more than once and is required only if the mode isn't none. Since this field authenticates the MX records without replacing them, the value doesn't have to resolve and the wildcard character * can be used as the leftmost label.

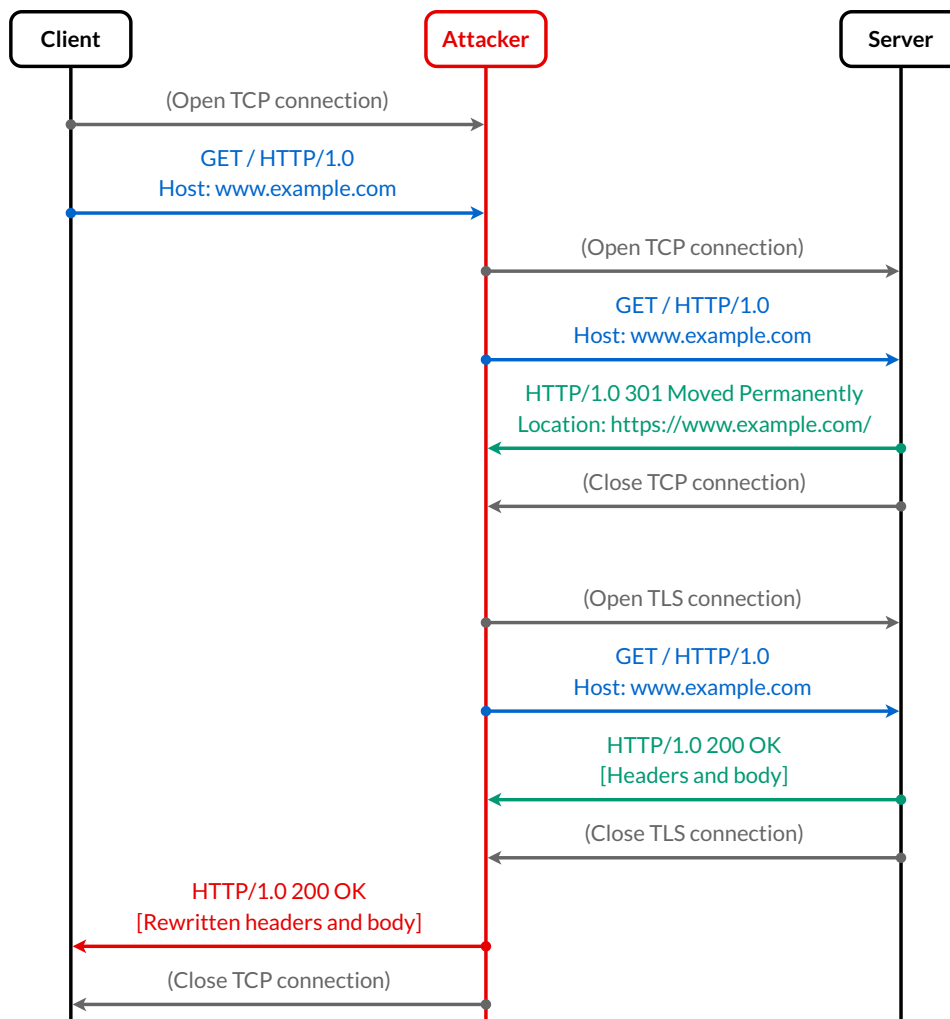
When an MTA-STS client connects to an ESMTP server with a valid policy, it must use the Server Name Indication (SNI) TLS extension with the name of the server as encountered in the verified MX record. Before we move on, here is an example policy:

```
version: STSv1
mode: enforce
mx: gmail-smtp-in.l.google.com
mx: *.gmail-smtp-in.l.google.com
max_age: 86400
```

The policy file at <https://mta-sts.gmail.com/.well-known/mta-sts.txt> in April 2021.

▼ HTTP Strict Transport Security (HSTS)

Opportunistic use of TLS is not unique to email. When a user enters a domain name into the address bar of their Web browser, the browser fetches the corresponding page via HTTP. Websites which support HTTPS, which is HTTP with Implicit TLS on port 443 instead of 80, usually redirect the browser to their secured version. An active attacker, however, can just strip the Location header field from the reply and deliver the rewritten content to the browser instead:



How a man in the middle can prevent the client from upgrading its TCP connection to TLS. If the attacker knows that the server redirects `http://www.example.com/` to `https://www.example.com/`, it can skip the first connection to the server, of course.

Whether to use Transport Layer Security (TLS) is encoded in the Uniform Resource Locator (URL). For example, if the user enters `https://www.example.com/`, the browser won't fall back to the insecure version at `http://www.example.com/`. On the other hand, if a browser tries to fetch `https://www.example.com/` when the user enters `www.example.com` and falls back to `http://www.example.com/` only if the attempt fails, the attacker can simply drop the browser's TLS connection as discussed above. This is different from email, where we don't have a mechanism to signal transport security policy in the address.

The problem of opportunistic security is addressed by HTTP Strict Transport Security (HSTS), which is specified in RFC 6797. HSTS allows Web servers to indicate with the Strict-Transport-Security HTTP response header field that they should be accessed only with HTTPS. This header field has a required max-age directive, which specifies the number of seconds for which the HSTS policy should be cached, and an optional includeSubDomains directive, which signals that the HSTS policy should be applied to subdomains of the current domain as well. A max-age value of zero indicates that the current domain opts out HSTS. The Strict-Transport-Security response header field is ignored in insecure HTTP responses. Here is an example:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

A Strict-Transport-Security response header field with a validity period of 365 days.

The problem with HSTS is that it doesn't protect the first connection to the server: An active attacker can prevent the browser from learning the server's HSTS policy. For this reason, browsers are delivered with an HSTS preload list. All domains on this list are accessed only via HTTPS. Chromium's HSTS preload list is 15 MB in size. All other browsers, for which we know how they source their HSTS preload list, rely on Chromium's HSTS preload list. You can submit your domain for inclusion in this list at hstspreload.org. Since it takes months to get a domain off this list again, the willingness to be included in this list has to be confirmed with a preload directive in the Strict-Transport-Security response header field. As far as I can tell, this directive has not been standardized in any RFC. The only reason why we need HSTS is backward compatibility. If you own a generic top-level domain, you can submit the whole top-level domain for inclusion in Chromium's HSTS preload list. Since GitHub Pages still doesn't support HSTS for custom domains, this blog has no Strict-Transport-Security header field and is on no preload list.

The following three security and privacy aspects of HSTS are worth mentioning:

- **Homograph attacks:** Just like all domain-based approaches, HSTS does not prevent homograph attacks. If a user enters a wrong address or visits a fraudulent link, an attacker can serve malicious content via HTTPS with a valid certificate for the fake domain or serve the content via HTTP given that the fake domain never used HSTS.
- **Network time attacks:** Unless a website is on the browser's HSTS preload list, its HSTS policy expires at some point. If an attacker can shift the time of the victim's computer into the future by attacking the Network Time Protocol (NTP), they can circumvent cached HSTS policies.
- **Web tracking:** Whether a domain uses HSTS is a piece of information which the browser stores and transmits to the server by accessing it via HTTPS instead of HTTP. By using many domains and enabling HSTS for a subset of them which is unique to each user, users can be tracked across websites even when cookies are disabled and private browsing is enabled.

▼ STARTTLS Policy List

Besides relying on earlier connections or an authenticated channel, transport security can also be required by the user or be configured by the administrator. An administrator can configure an outgoing mail server to require TLS for specific recipient domains. In order to decrease the effort for administrators of outgoing mail servers, the Electronic Frontier Foundation (EFF) published the STARTTLS Policy List from 2014 to 2020. While I understand that maintaining such a project is a lot of work, the STARTTLS Policy List could still serve as an MTA-STS preload list similar to HSTS preload lists in order to prevent downgrade attacks on connections where the client has no cached MTA-STS policy. Since its discontinuation, no such policy list exists.

SMTP TLS Reporting (TLSRPT)

SMTP TLS Reporting (TLSRPT) is specified in RFC 8460. With it, domain owners can ask sending mail servers to report transport security failures to them, which allows them to detect misconfigurations and attacks. If you're certain that all emails are still being delivered to you, you're much more likely to enforce strict transport security. Just like DMARC reporting, TLSRPT uses a DNS record to specify the endpoints to which reports should be sent once a day.

The following tool queries the TLSRPT record with Google's DNS API and checks its format with a regular expression:

Domain:   

▼ TLSRPT DNS record

The endpoints for the report are specified in a TXT record at `_smtp._tls.{RecipientDomain}`. The record has two fields:

- `v`: The version of the TLSRPT standard. This field has to come first and the only supported value is TLSRPTv1 at the moment.
- `rua`: A comma-separated list of Uniform Resource Identifiers (URIs) as endpoints. The following two schemes are supported:
 - `mailto`: The report is sent to the specified email address. The email must have a valid DKIM signature and the service type of the DKIM key should include TLSRPT. Any TLS-related failures must be ignored when delivering the report. Unlike DMARC, recipients with a different domain don't have to approve TLSRPT reports since spam is less likely.
 - `https`: The report is submitted via a POST request to the specified Web address. Certificate validation errors may be ignored when submitting the report. Since the submitter is not authenticated, I advise you not to use this method. I have no idea why this shortcoming isn't mentioned in the security considerations.

The fields are separated by a semicolon and the keys and values are separated by an equals sign. Here is an example record:

```
v=TLSRPTv1;rua=mailto:sts-reports@google.com
```

The TXT record at `_smtp._tls.gmail.com`. Whitespace is allowed before and after semicolons.

▼ TLSRPT report format

TLSRPT reports are formatted with the JavaScript Object Notation (JSON) according to this schema. The reports should be compressed with GZIP both when sent as an attachment via email and when submitted via HTTPS. The filename of the report should be `{SenderDomain}!{RecipientDomain}!{BeginTime}!{EndTime}[!{UniqueId}].json[.gz]` and the subject of the email `Report Domain: {RecipientDomain} Submitter: {SenderDomain} Report-ID: {ReportId}`, with all dates in Unix time. In case of failures, the report indicates the cause. Here is an example report, which I've formatted for better readability:

```
{
  "organization-name": "Google Inc.",
  "date-range": {
    "start-datetime": "2021-04-09T00:00:00Z",
    "end-datetime": "2021-04-09T23:59:59Z"
  },
  "contact-info": "smtp-tls-reporting@google.com",
  "report-id": "2021-04-09T00:00:00Z_ef1p.com",
  "policies": [
    {
      "policy": {
        "policy-type": "no-policy-found",
        "policy-domain": "ef1p.com"
      },
      "summary": {
        "total-successful-session-count": 1,
        "total-failure-session-count": 0
      }
    }
  ]
}
```

A report which I've received from Google with the filename `google.com!ef1p.com!1617926400!1618012799!001.json.gz` in an email with the subject `Report Domain: ef1p.com Submitter: google.com Report-ID: <2021.04.09T00.00.00Z+ef1p.com@google.com>`. I'm not sure what `total-successful-session-count` means if the `policy-type` is `no-policy-found`. Would it count as a failure if TLS cannot be negotiated? Or are all sessions successful if no policy was found? You find an example report with failure details in [RFC 8460](#).

▼ TLSRPT report conditions

TLSRPT can be used with [MTA-STS](#) and with [DANE](#). [RFC 8461](#) specifies which MTA-STS failures shall be reported:

- Failure to fetch the [policy file](#) via HTTPS when a valid [DNS record](#) was found.
- Failure to refresh a cached policy which is still valid unless its mode is none.
- Failure to negotiate TLS with a valid server certificate when delivering a message.

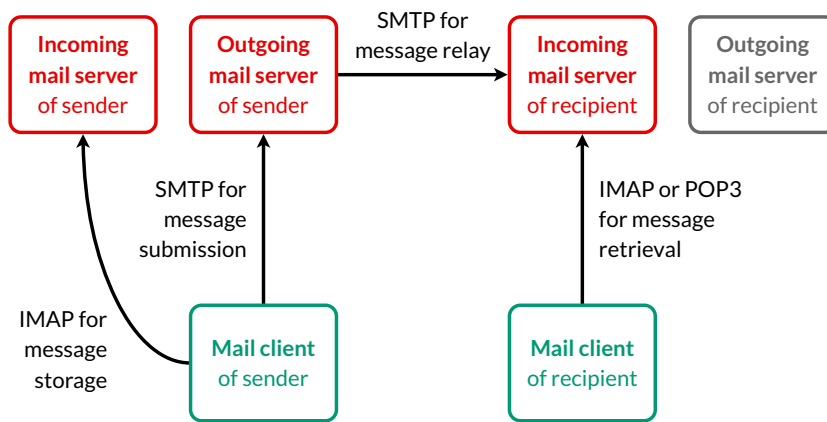
[RFC 8460](#) adds invalid policies to this list. In the case of DANE, [RFC 8460](#) mentions invalid [TLSA records](#) and no valid [DNSSEC signatures](#) as reportable failures. [Other result types](#) include missing support for [STARTTLS](#) and certificates which expired, were not trusted, or had [no matching name](#).

▼ Reporting in header fields

Unfortunately, you cannot see in the [raw message](#) whether the outgoing mail server of the sender enforced [transport security](#). The introduction of a [Transmitted header field](#) to let the client record the checks it performed was once considered but then [dropped again](#) in a [later revision](#). A client-side equivalent to the [Received header field](#) thus never made into the [list of message header fields](#). Such a header field would only make sense if it was covered by a [DKIM signature](#). Otherwise, a [man in the middle](#) can simply replace it, which prevents the recipient from detecting whether a man in the middle was present or not. Since a man in the middle can encrypt its communication with the incoming mail server of the recipient, a Received header field with a [with clause](#) of ESMTPS ([ESMTP](#) with [STARTTLS](#)) also provides less assurance than one would hope.

End-to-end security

Instead of relying on mail servers to perform [domain authentication](#) and enforce [transport security](#), senders and recipients can take matters into their own hands and secure their communication themselves. This idea is often referred to as [end-to-end encryption \(E2EE\)](#). Since protecting the authenticity of the content is usually just as important as protecting its confidentiality, I prefer the term end-to-end security. As we've seen [earlier](#), arbitrary content can be sent via email. As long as the sender and the recipient agree on which cryptographic algorithms and which encoding they want to use, they can use any technique they want, such as [one-time pad encryption](#) combined with a [message authentication code \(MAC\)](#). While end-to-end security doesn't have to be [standardized](#), doing so is valuable for two reasons: The more people use the same technique, the more useful it becomes for each user, which is known as a [network effect](#), and if everyone uses the same technique, it can be integrated into mail clients, which makes it easier to use.



If the mail client of the sender encrypts and authenticates the message for the mail client of the recipient, none of the mail servers have to be trusted (beyond delivering or storing the message).

End-to-end security has two advantages:

- **No trust in mail servers required:** In theory, if you don't trust any email service provider, you can just host your emails yourself. In practice, however, running your own mail server on your own hardware is a hassle. Beyond the technical complexity of running a mail server, you may want to share the infrastructure with other users in order to reduce costs. Without end-to-end security, everyone has to trust the administrator of the mail servers. If you employ end-to-end security on all your messages, you can choose any free email service provider who delivers your messages reliably.
- **Secrets on clients instead of servers:** In order to receive emails from anyone, incoming mail servers have to be reachable from anywhere on the Internet. If a security hole is found in the used software, mail servers become vulnerable immediately. Given that servers are typically shared by many users, they are a prime target for attacks. If you employ end-to-end security with all your contacts, an attacker who compromised your mail server can neither read your communication nor send messages in your name. If, on the other hand, your mail client or your computer is compromised, it's over with and without end-to-end security.

End-to-end security also has some disadvantages:

- **No remote search:** Your mail client has to store all messages locally if you want to be able to search for a message based on its content. Without end-to-end encryption, your mail client can ask your incoming mail server to perform the search using the SEARCH command of IMAP. While storing all messages locally is no longer a problem for modern smartphones, it might still be one for smartwatches. End-to-end security requires thick clients instead of thin clients.
- **No partial downloads:** IMAP allows clients to fetch just certain parts of multipart messages. Since the body of a message is usually signed and encrypted as a single unit, bandwidth-constrained mail clients cannot download the text of an end-to-end secured message without its attachments.
- **Archiving of messages:** If you lose your decryption key, you can no longer access your messages (unless your mail client stores them in plaintext on your computer). While this can be an annoyance for individuals, it can be a real problem for companies, who must archive their electronic communication in order to avert spoliation of potential evidence.
- **Message filtering:** Mail servers cannot scan encrypted messages for malware or discard them as spam based on their content.

There are two main standards for end-to-end security: Secure/Multipurpose Internet Mail Extensions (S/MIME) and Pretty Good Privacy (PGP), which was standardized as OpenPGP. Unlike the other fixes in this chapter, both of them have existed since the 90s. The main difference between them is how public keys are authenticated, distributed, and revoked. Otherwise, they are quite similar.

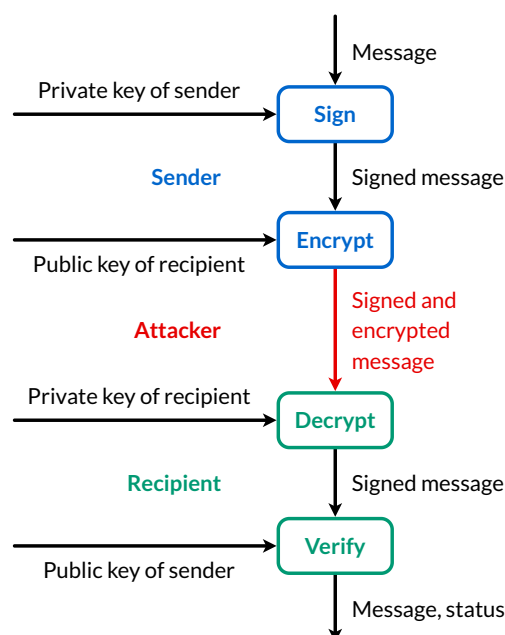
Aspect	<u>Secure/Multipurpose Internet Mail Extensions (S/MIME)</u>	<u>Pretty Good Privacy (PGP)</u>
Standards	<u>RFC 8551</u> : S/MIME formats <u>RFC 8550</u> : Certificate handling <u>RFC 5652</u> : Message syntax <u>RFC 3370</u> : Algorithms	<u>RFC 4880</u> : OpenPGP formats <u>RFC 3156</u> : Content types
Certificate format	<u>RFC 5280</u> : <u>X.509</u>	<u>Specific to OpenPGP</u>
Public-key binding	<u>Certification authorities</u>	<u>Web of trust</u>
Public-key distribution (before relying on <u>DNSSEC</u>)	Attached to the message as part of the signature or through an <u>internal directory</u> .	Public <u>key server</u> , personal website, or <u>Autocrypt header field</u>

Aspect	<u>Secure/Multipurpose Internet Mail Extensions (S/MIME)</u>	<u>Pretty Good Privacy (PGP)</u>
Public-key revocation (before relying on DNSSEC)	Certificate Revocation Lists (CRL) or Online Certificate Status Protocol (OCSP) by the issuing certification authority	Key revocation signature by the owner of the key
DANE resource record type	SMIMEA	OPENPGPKEY
Content type for encryption	application/pkcs7-mime	application/pgp-encrypted
Content type for signature	application/pkcs7-signature	application/pgp-signature
Primary user group	Business world	Security specialists
Costs for users	You have to pay for the certificate but there are free offers for personal use	None

Comparison of [Secure/Multipurpose Internet Mail Extensions \(S/MIME\)](#) and [Pretty Good Privacy \(PGP\)](#).

▼ Modes of operation

S/MIME and PGP support the same three modes of operation: You can either sign a message, sign and encrypt a message, or only encrypt a message for its recipients without signing it. When doing both, the message is first signed and then encrypted:



How a message is signed and encrypted by the sender and then decrypted and verified by the recipient. If you don't know the public key of the recipient, you can only sign your message.

If a message is first encrypted and then signed, an attacker can attach their own signature to the [ciphertext](#), thereby claiming the message for themselves without knowing its content. This is a problem whenever the message refers to the sender's public key. For example, if Alice informs Bob via email that she uses a new signing key and includes a [shared secret](#) in her message to authenticate herself, the attacker can replace the signature and impersonate Alice in future communication with Bob without having to learn the shared secret. If a [cryptographic protocol](#) is not constructed carefully, signing ciphertexts can also fail in a [different way](#). Please note that in the case of [message authentication codes \(MACs\)](#), you should first encrypt the message and then authenticate the ciphertext in order to prevent [chosen-ciphertext attacks](#), such as [padding-oracle attacks](#).

▼ Deniable authentication

[Digital signatures](#) entail [non-repudiation](#). As discussed [earlier](#), this is not always desirable. There are three different properties:

- **Integrity:** An attacker cannot alter the [ciphertext](#). Unless you want [homomorphic encryption](#), [ciphertext malleability](#) is never desirable. We're not interested in the integrity of [plaintexts](#) here because we want more than just [error detection](#).
- **Authenticity:** The recipient can authenticate the sender of the message.

- **Non-repudiation:** The sender cannot deny having sent the message.

While non-repudiation ensures authenticity and authenticity ensures integrity, integrity does not entail authenticity and authenticity does not entail non-repudiation. Integrity is useful because it allows the sender to achieve authenticity with the content of the message. For example, when Alice sends an encrypted message to Bob and Bob includes the original message in his encrypted response, Alice knows that the response came from Bob because only Bob (and herself) had access to her original message. Without integrity protection, an attacker might have altered Bob's response or Alice's original message. Integrity can be achieved by appending a checksum of the plaintext to the plaintext before encrypting both. If the checksum is no longer valid after decryption, the recipient knows that someone tampered with the ciphertext. Non-repudiation is achieved with digital signatures, where only the signer can generate the signature but everyone can verify the signature. Authenticity without non-repudiation is accomplished with a message authentication code (MAC), which require that the sender and the recipient have a shared secret. A message authentication code can be verified only by those who know the shared secret and everyone who can verify a message authentication code can also generate it. As a consequence, the recipient cannot prove to a third party that a message was sent by the sender. This feature is important for off-the-record messaging and it can also be achieved with designated verifier signatures. Here is a summary of which mechanism provides which of the above properties:

Property	Message authentication code (MAC)		
	Checksum	Message authentication code (MAC)	Digital signature
Integrity	✓	✓	✓
Authenticity	✗	✓	✓
Non-repudiation	✗	✗	✓

Which mechanism ensures which properties. More is not always better.

As we've seen in the previous box, both S/MIME and PGP support digital signatures. Ciphertext integrity has been added to S/MIME with the Authenticated-Enveloped-Data content type as specified in RFC 5083 (other than what its name suggests, this content type does not provide authentication) and to PGP with the Modification Detection Code (MDC), which consists of a SHA-1 hash appended to the message before encryption. PGP does not support deniable authentication but you can share a new private key with the recipient or use symmetric-key encryption with a shared key. S/MIME doesn't really support deniable authentication either. You can use a non-interactive Diffie-Hellman key exchange to encrypt the content-encryption key for each recipient. RFC 6278 even defines how to use elliptic-curve Diffie-Hellman (ECDH) where the sender's and the recipient's key pair are static instead of ephemeral. However, this approach cannot be used when a message is sent to more than one recipient because the common content-encryption key is not bound to this set of recipients and could be used by a recipient to frame the sender in a private conversation with another recipient. Even the Authenticated-Data content type shares a single authentication key with all recipients instead of appending a different message authentication code for each recipient.

▼ Compression before encryption

Both S/MIME and PGP support compression. A corresponding content type to use compression alone in a message exists only for S/MIME, though. GNU Privacy Guard (GnuPG or GPG), which is the most popular open-source implementation of the OpenPGP standard, compresses by default before encryption unless it detects that the data is already compressed. If you use GPG to encrypt computer-generated text such as reports or notifications, you should disable compression with -z 0 in order not to leak how similar the dynamic part of your message is to the static part. Encryption does not conceal the length of the input and compression produces a shorter output when more parts of the message are the same. If an attacker can influence the dynamic part of the message, for example by triggering the generated notification, your system is vulnerable to a so-called compression-oracle attack, which is an adaptive chosen-plaintext attack.

▼ Multipart message nesting

When used to sign and encrypt emails, S/MIME and PGP are applied to MIME body parts. Any part in the tree of a multipart message can be signed or encrypted but S/MIME and PGP are usually applied to the whole body of a message. The output is itself a MIME part with a content type of application/pkcs7-mime, multipart/signed, or multipart/encrypted. Given that the output is just another MIME part, the operations can be applied in any order. Therefore, a message can be first encrypted and then signed but as we discussed, this is not recommended. The flexibility of multipart nesting leads to a lot of complexity and complexity is detrimental to security, which is not ideal for end-to-end security. Security researchers published several attacks on S/MIME and PGP in 2018. Among other things, they showed that several mail clients, including Apple Mail and Thunderbird, allowed HTML content to span several MIME parts. An attacker could simply wrap an encrypted part with `` in order to trick mail clients into sending the decryption to them. A vulnerability like this is yet another reason to disable remote content in your mail client. Let's look at some example messages:


```
MIME-Version: 1.0
Content-Type: application/pkcs7-mime; smime-type=signed-data; name=smime.p7m
Content-Disposition: attachment; filename=smime.p7m
Content-Transfer-Encoding: base64
```

[Base64-encoded content and signature]

S/MIME signature using the application/pkcs7-mime format. [↗](#)

```
MIME-Version: 1.0
Content-Type: multipart/signed; micalg=sha-256;
  protocol="application/pkcs7-signature";
  boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: text/plain; charset=us-ascii

This part is signed.

--UniqueBoundary
Content-Type: application/pkcs7-signature; name=smime.p7s
Content-Disposition: attachment; filename=smime.p7s
Content-Transfer-Encoding: base64
```

[Base64-encoded signature with metadata]

--UniqueBoundary--

S/MIME signature using the multipart/signed format. [↗](#) micalg stands for message integrity check (MIC) algorithm.
The advantage of this format is that users can read the message even if their mail client doesn't support S/MIME.

```
MIME-Version: 1.0
Content-Type: application/pkcs7-mime; smime-type=authEnveloped-data; name=smime.p7m
Content-Disposition: attachment; filename=smime.p7m
Content-Transfer-Encoding: base64
```

[Base64-encoded ciphertext with metadata]

S/MIME encryption with integrity protection. [↗](#)

```
MIME-Version: 1.0
Content-Type: multipart/signed; micalg=sha-256;
  protocol="application/pgp-signature";
  boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: text/plain; charset=us-ascii

This part is signed.

--UniqueBoundary
Content-Type: application/pgp-signature; name=signature.asc
Content-Disposition: attachment; filename=signature.asc
```

-----BEGIN PGP MESSAGE-----

[Base64-encoded signature with metadata]
=[Base64-encoded checksum of the signature]
-----END PGP MESSAGE-----

--UniqueBoundary--

PGP signature with the message in the first part. [↗](#) The checksum is a 24-bit cyclic redundancy check (CRC). Some implementations use BEGIN PGP SIGNATURE instead of BEGIN PGP MESSAGE.

```

MIME-Version: 1.0
Content-Type: multipart/encrypted;
    protocol="application/pgp-encrypted";
    boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: application/pgp-encrypted

Version: 1

--UniqueBoundary
Content-Type: application/octet-stream; name=encrypted.asc
Content-Disposition: attachment; filename=encrypted.asc

-----BEGIN PGP MESSAGE-----

[Base64-encoded ciphertext with metadata]
=[Base64-encoded checksum of the ciphertext]
-----END PGP MESSAGE-----

--UniqueBoundary--

```

PGP encryption with metadata in the first part. 2 If the plaintext has been signed, you get the format of the previous example without `MIME-Version: 1.0` after decryption.

▼ Securing header fields

An email consists of **header fields** and a **body**. By transforming **MIME parts**, S/MIME and PGP can protect only the body of a message but not its header fields. If you sign and/or encrypt a message, you want to sign and/or encrypt the **subject** as well. The **recipients** should also be covered by the signature. Otherwise, one of the recipients can forward the signed body to someone else. For example, when Alice sends “I have to cancel our meeting” to Bob, Bob shouldn’t be able to forward the signed body to Carol. There are three ways to secure header fields. Before you rely on end-to-end security, make sure that your mail client uses one of them.

- [RFC 8551](#) suggests that mail clients can wrap the full message with a **content type** of `message/rfc822`. The receiving mail client has to decide what to do when the protected header fields diverge from the unprotected header fields.
- [RFC 7508](#) specifies how a sending mail client can include header fields in the signed attributes of its S/MIME signature.
- [This draft](#) specifies how header fields can be repeated in the signed MIME part with a content-type parameter of `protected-headers="v1"`. If the message is encrypted, the original subject should be replaced with three periods.

Thunderbird follows the last specification. A message with protected header fields looks as follows after decrypting it:

```

From: Alice <alice@example.org>
To: Bob <bob@example.com>
Subject: ...
Date: Wed, 21 Apr 2021 10:55:18 +0200
Message-ID: <unique-identifier@example.org>
MIME-Version: 1.0
Content-Type: multipart/signed; micalg=sha-256;
    protocol="application/pgp-signature";
    boundary="UniqueBoundary"

--UniqueBoundary
Content-Type: text/plain; charset=us-ascii; protected-headers="v1"
From: Alice <alice@example.org>
To: Bob <bob@example.com>
Subject: Our secret plan
Date: Wed, 21 Apr 2021 10:55:18 +0200
Message-ID: <unique-identifier@example.org>

This part is signed (including the header fields).

--UniqueBoundary
Content-Type: application/pgp-signature

-----BEGIN PGP SIGNATURE-----

```

```
[Base64-encoded signature with metadata]
=[Base64-encoded checksum of the signature]
-----END PGP SIGNATURE-----

--UniqueBoundary--
```

PGP signature with protected header fields. [↗](#) The original subject is replaced with three dots only when the message is encrypted.

▼ SMIMEA resource record

A challenge when using end-to-end security is to get the public key of the recipient and to verify the public key of the sender. [RFC 8162](#) specifies an experimental standard to anchor S/MIME certificates in the DNS just like [DANE](#). For this purpose, it defines the [SMIMEA record type](#), which has the same format and semantics as the [TLSA record type](#). All four [certificate usages can be used](#) and, unlike DANE, publishing the full certificate in the SMIMEA record is not discouraged. As a sender, you need the unhashed public key of the recipient in order to encrypt the message for them. As a recipient, it's enough if you can verify the public key of the sender with its hash in the SMIMEA record.

The [location of the SMIMEA record](#) is determined by hashing the [UTF-8-encoded local part](#) of the email address after removing any [comments and folding whitespace](#) and [enclosing double quotes](#). The local part should be [NFC normalized](#) before hashing it with [SHA-256](#). The [hexadecimal encoding](#) of the first 28 of the 32 bytes of the hash is used to form the domain name as follows: {Hash}._smimecert.{Domain}. The local part is hashed in order to support [internationalized email addresses](#). A problem of hashing is that provider-specific [normalizations of the local part](#), such as lowercasing all characters and removing all periods, [cannot be considered](#). Users should use the email address as written in the S/MIME certificate or PGP identity or publish a SMIMEA record for all common variants of their local part. Instead of publishing a separate record for each user, an organization can publish a [wildcard record](#) with a trust anchor which issues certificates for all its members. This approach doesn't allow senders to look up the public keys of members of this organization, though. In order to reduce the load on DNS caches, such a wildcard record should be a small CNAME record which points to the much larger SMIMEA record at a single domain name.

When it comes to privacy, publishing information about individual users in the DNS is not ideal. Unless a client uses [DNS over TLS](#), queries for SMIMEA records can be monitored by anyone in the user's network. Additionally, SMIMEA records [have to be authenticated with DNSSEC](#). If a domain uses NSEC instead of NSEC3 records for authenticated denial of existence, anyone can [walk the zone](#) to determine the hash of all local parts with a SMIMEA record. Since the local part is hashed without the domain part of the email address, short and common local parts [can be found](#) with a single [rainbow table](#) across domains.

The following tool queries the SMIMEA record of an email address with [Google's DNS API](#). Besides [Unicode normalization](#), the local part is hashed as is. You can test the tool with the email address `ietf-dane-phil@spodhuis.org`, which I found [here](#).

Email address:   

▼ OPENPGPKEY resource record

[RFC 7929](#) specifies how users can publish their OpenPGP keys in the DNS with the new [OPENPGPKEY record type](#). OPENPGPKEY resource records are [transmitted in binary](#) but [presented in Base64](#). Even though OPENPGPKEY records [must be authenticated with DNSSEC](#), a retrieved key must still be verified by the user out-of-band. The [location of OPENPGPKEY records](#) is determined in the same way as it is done for [SMIMEA records](#) with the exception that the subdomain is `_openpgpkey` instead of `_smimecert`. Indirections with CNAME records are allowed as long as the original email address is listed as one of the user identities in the found OpenPGP key. A user identity can have the [wildcard character *](#) as its local part so that the key can be used for [all addresses of the given domain](#). Besides [obtaining the key for an email address](#), OPENPGPKEY records allow others to detect [when a key has been revoked](#).

The following tool queries the OPENPGPKEY record of an email address with [Google's DNS API](#). You can test the tool with the email address `security@ef1p.com`. The [key's fingerprint](#) is `7044 3D35 13F7 9AD9 2527 667F 6B14 3BF1 470C 9367`. You can use this key if you want to report security-related issues to me. The tool displays the key in [ASCII armor](#). The first two lines and the last two lines are not part of the OPENPGPKEY record. (The second last line is a [24-bit checksum](#).)

Email address:   

If you want to publish an OPENPGPKEY record, you can export your key with one of the following two commands depending on whether your [domain name registrar](#) supports the OPENPGPKEY presentation format (consult [--export-options](#) for details):

```
# Export the public key of the given user in the OPENPGPKEY presentation format:
$ gpg --export --export-options export-minimal,no-export-attributes security@ef1p.com | base64

# Export the public key of the given user in the generic record syntax of RFC 3597:
```

```
$ gpg --export --export-options export-minimal,no-export-attributes,export-dane security@ef1p.com
```

You can configure GnuPG to look for OPENPGPKEY records automatically with the `--auto-key-locate` command.

▼ SSHFP resource record

After having seen how public keys can be published in the DNS for [TLS](#), [S/MIME](#), and [OpenPGP](#), I just want to mention that this is also supported for the [Secure Shell Protocol \(SSH\)](#), which has nothing to do with email. [RFC 4255](#) specifies the [SSHFP record type](#), which consists of the following three fields:

- **Algorithm number:** The algorithm of the SSH key. IANA maintains the [registry of assigned values](#).
- **Fingerprint type:** The [hash function](#) used to determine the key's fingerprint. IANA maintains the [registry of assigned values](#).
- **Fingerprint:** The [fingerprint](#) of the SSH key. The fingerprint is transmitted in binary and presented with [hexadecimal digits](#).

When a user connects to an SSH server for the first time, they need to confirm the authenticity of the server's public key. Accepted public keys are added to the file `~/.ssh/known_hosts` and used to authenticate future connections to the same server. While the user should verify the server's public key out-of-band, the presented key is often accepted without any verification. This model is known as [trust on first use \(TOFU\)](#): If the first connection was secure, all future connections will be secure. If the first connection was intercepted by a [man in the middle](#), then all future connections will be compromised as well.

SSHFP records leverage the authentication provided by [DNSSEC](#) to secure the initial connection to an SSH server. They are published at the domain of the server and you can look them up with the [DNS resolver](#) from the first article. For example, the domain `ccczh.ch` has two SSHFP records and the server's public key has to match at least [one of them](#). [OpenSSH](#) allows you to generate the SSHFP records with the [following command](#) on the server: `ssh-keygen -r {Hostname}`. If your [name server](#) does not support the [SSHFP presentation format](#), you can use the `-g` option to generate the DNS records in the [generic format](#). In order to use SSHFP records to verify a server's public key, you have to set the [VerifyHostKeyDNS configuration](#) of your SSH client to yes or ask. You can do this for a single connection with the `-o` option: `ssh -o "VerifyHostKeyDNS ask"`. You can also set this configuration for all hosts in the system-wide settings at `/etc/ssh/ssh_config` or the user settings at `~/.ssh/config`:

```
Host *  
    VerifyHostKeyDNS ask
```

If no such folder or file exists in your user directory, you can create them with:

```
$ mkdir -p ~/.ssh  
$ chmod 700 ~/.ssh  
$ touch ~/.ssh/config  
$ chmod 600 ~/.ssh/config
```

The copyright of this article and its graphics belong to Kaspar Etter. You can share this article in any form as long as you give proper attribution.