No one can force me to have a secure website!!!

Dr. Tom Murphy VII, Ph.D.
March 2026

I have one of the oldest websites on the internet,[1] dating back to at least AD 1996. It has so far outlived its original host (AOL Hometown) by 18 years. Now that 99.9999999% of the internet is deepfake pornography, AI-generated cryptocurrency scams, and Italian Brainrot, a website being in Lycos's "Top 5% Of The Web" is a triviality achieved at the first human keystroke. But in the late 1990s, it used to *mean* something.

Here's the thing: I just want my public website to be on the internet and accessible in a web browser!

I don't want you to have to log in, or give me permission for cookies, or enable notifications, or it's better in our app, or sign up for the e-mail list, or figure out which one is the search box and which one is a trick to get me to enter my search terms in the e-mail sign up box (??), nor dismiss 15% off coupon / privacy policy pop-up / how may I help you? AI chat assistant, nor do I want you to consent to or refuse consent to or leave open in silent protest data collection per the EU General Data Protection Regulation, click to enable Flash Player, or Flash Player no longer supported, or 15 second interstitial video advertisement, or ☑ W3C XHTML compliant.[1]

U N F O R T U N A T E L Y, the Internet is not that place any more! One possibly controversial opinion of mine is that one contributor to this enshittification[2] of the web is the phenomenon I shall call "toxic max-security." This is where security defaults are turned up to a silly degree, scaring users, creating pointless toil and chores for good guys, and arguably *reducing* the usefulness of security due to alarm fatigue.[3] The specific toxicity that has set me off in this here SIGBOVIK paper is this ⚠ RED ALERT:

Your connection is not private

Attackers might be trying to steal your information from **tom7.org** (for example, passwords, messages, or credit cards). Learn more about this warning

NET::ERR_CONNECTION_REFUSED

Advanced Back to safety

This is a real-life warning screen shown to me when I tried to use the Chrome "web browser" to visit my own website at `http://tom7.org/`. My website has never had HTTPS, does not give any indication that it wants or expects HTTPS. I did not ask for the `https://` protocol in the URL. The website is completely public, available on the Internet Archive, and does not process or contain any private information or anything sensitive in any way. Its privacy policy is: LOL. I can tell you for certain that when I was connecting to my own website, there was no attacker trying to "steal my credit card." First of all, need we really discuss the obvious distinction between stealing a credit card and stealing a credit card *number*? Second, there is simply no place to even put my credit card number on my website short of posting it as a comment on my blog, which, be my guest...!

In order to get to my website, the user must click on "Advanced" and endure a further mansplaination and then click the tiny text that says "Proceed to tom7.org (unsafe)".

Having seen this screen, I officially declare that I have 0xBEEF5 with someone on the Chrome product team who is preventing users from getting to my website for the sake of toxic max-security. But before that final straw, there were many other irritations.

---

1. Find it at http://tom7.org/

Earlier straws:

• I want to go to an old good website, but I can't, because their certificate has actually expired. Why is this even a problem? This is like when I go to buy a beer and the guy sends me away because my driver's license expired. Do you think that I GOT YOUNGER in the time since it was valid?
• I try to copy-paste a URL from my site but Chrome "helpfully" turns the `http:` prefix into `https:`, making it no longer work.
• I try to go to an HTTP site in incognito mode, perhaps precisely because I don't really trust it and don't want it sending cookies or infesting me with cookies. Chrome refuses on the grounds that when I'm in incognito mode, I must require encryption (?) even though that's not at all what incognito mode is about (??).
• I want to get on the janky airplane wi-fi, but this requires attempting to load any website over HTTP so that the router replaces the page with the wi-fi login page. But the browser keeps automatically trying to go to the HTTPS version.
• I just want to do some software development but `profiler.firefox.com` is served over HTTPS and so it can't load the profile from an HTTP server (mixed-content), but this is on a development computer while I'm on vacation and I just don't have a domain for it, and so HTTPS is not even possible!

Now I do want to be clear: I am glad that we have `https`, and I think it should be supported widely, and certainly for my bank account and e-mail and moderately popular YouTube channel. I think it should be clear to a user who is capable of paying attention when they might not be on the site they think they're on. I truly think cryptography is fascinating and I liked when there was a little green lock in my URL bar. But in this paper I hope to convince you that `https` has lost the plot somewhat, and will point out some of its own shortcomings and hypocrisies. I'll show an alternative approach to software development that is more emotionally open to its users.

Regarding my new enemy, I think this person probably thinks they are one of the good guys. But I propose that they are more like the following chumps:

• The person that added signing requirements for programs on Mac OS, so that users get a scary warning if they download a program that a hobbyist made. By the way, to sign a piece of software, you need to pay an extortive $100/year fee to Apple.[4]
• The tool that added an RFID scanner to my expensive General Electric brand (do not recommend) refrigerator (presumably increasing its cost). It only scans an RFID tag in the $60 water filter cartridges (presumably increasing their cost) and prevents you from using one that does not have the proper Drinking Rights Management credentials. By the way this scanner is pretty janky and sometimes just doesn't work.
• The turd that requires my video game to install kernel-mode anti-cheat provisions that accuse me of cheating after installing an official Windows update, and that requires me to be online to play the *single-player mode* of a video game, and which will eventually be completely unplayable when they turn down the server.
• The absolute shits that have locked down corporate computers with the assumption that the user can't have a legitimate reason to change settings on it, put in a USB stick, use the command line, run an "untrusted" application like `emacs` or something that I just wrote and compiled myself, or basically any application other than a web browser, even if that user has been programming for 40 years and has a Ph.D. in computer science and was hired for that very experience.

This slander applies not just to the product managers who instigate such user-hostile ideas, but to the engineers who implement them without questioning the impact on the user. Or *especially* if the engineer disagrees with the idea but does it anyway because they have abdicated their responsibility to follow the direction their moral compass points. Grow some guts! The system is actually not set up very well to force you to do things like this.

But don't worry: This paper is not only an extended, petty rant about a time that I was annoyed by a warning message. It is also an extended, petty project to attempt to get some kind of symbolic revenge for the time that I was annoyed by a warning message.[2]

TLS

"TLS" is the new name for SSL, which is the thing that makes the letter S in HTTPS. It uses cryptography for two purposes here: To ensure that you are talking directly to the site you think you are, and to encrypt the communication so that observers

can't read it.

Encryption is pretty easy. There are loads of algorithms that are fast and that provide adequate security.[3] In this paper we will use AES-256.[6] To send encrypted data back and forth, you need to agree on what encryption keys to use. There are various approaches, but the one relevant to this paper is just to use public-key cryptography. It works like this: When you connect to the server (before anything is encrypted), it offers up its public key. The public key can be used to send encrypted messages that only the owner of that key can decrypt (using the private key). The client can then decide on an AES key, encrypt it with the server's public key, and send it to the server.

Oops—HACKED!! This doesn't work, though. While a passive observer can't uncover the key, an attacker can insinuate themselves into the connection. To the client, they pretend to be the server, passing their *own* public key to the client and receiving the client's chosen AES key. To the server, they pretend to be the client, negotiating a secure connection and forwarding data between the real client and the server. This "man-in-the-middle" (often a computer and not a biological man) is invisible to the client and server, but can read (and often modify) any data encrypted data sent between them. The possibility of such a rude intermezzo is arguably the core difficulty of designing protocols like TLS.

The way that TLS prevents against man-in-the-middle attacks is for the client to check that the public key actually belongs to the server. This is accomplished by having someone *else* that the user absolutely trusts vouch for the key that the server presents. This Certificate Authority does this by "signing" the key–domain pair using their own private key, creating a *certificate*. Your browser can verify this using the Authority's corresponding public key, which just comes with your browser.[4]

TLS technically supports self-signed certificates, where the certificate is signed with the key itself ("I am me!" —*Hacker*). No browsers allow self-signed certificates because the man-in-the-middle can just make them too. So in order to have a website on the internet that users can access, you must submit to a Certificate Authority.

Fundamentally, a Certificate Authority's job is simply to vouch that a specific public key belongs to a specific domain name. Their voucher is a time-limited certificate[7] that is presented by the server when you connect to it. So any server that wants to use `https` needs to regularly get a certificate from a CA for each of their key–domain pairs. They're supposed to check that the key actually belongs to the domain name; originally this happened through some kind of verification process, like when you try to go to the deepfake pornography web site but it demands a credit card number to check that you're really 18 years old.

Let's Encrypt

So in the first decade of the 2000s, if you wanted to have a 🔒 green padlock icon on your site, you needed to find some security professionals who would pretend to do some Military Grade secrecy to keep their numbers secret, and vouch that you had paid them the extortion fee. Despite the proptosis-inducing invoices, all their data center laser grid security systems and tactical sunglasses, many CAs still issued many improper certificates. Trusted Certificate Authorities such as Thawte, StartCom, Comodo, DigiNotar, TurkTrust, NICCA, CNNIC, WoSign, Symantec, and Certinomis all shit the bed,

---

2. In fairness, in the months since launching my petty project (and for what I assume are unrelated reasons) Chrome has mostly retreated on its red alerts. I would say that I still have `0xBEEF5` and I assume that someone out there is just itching to do it again, but I would describe its behavior as of SIGBOVIK 2026 as "medium straws" and not "last straw."

3. Take even a last-century classic like 3DES, which OpenSSL declares a "weak cipher." This one is weak mainly in the sense that if you capture about 785 Gigabytes of traffic encrypted with the same key, *and* you are able to know what a lot of that traffic already is, then you can probably decrypt some blocks simply because you've likely induced a collision between 64-bit blocks due to the birthday paradox.[5] *All* 64-bit ciphers in CBC mode have this issue, like all ciphers with 40-bit keys are weak to brute force attacks.

4. SSH isn't quite TLS but it needs to solve a similar problem. Absent certificates (which is typical) it just gives you a message like

```
The authenticity of host 'github.com (140.82.113.4)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

... which is a perfectly reasonable compromise (hehe) at the command line. Once you connect this way it just remembers that key, which is sometimes called "TOFU" (Trust on First Use).

with the malfeasance warranting different levels of spit-take.[8] The typical mistake is failing their one job: Checking that the key belongs to the domain before issuing the certificate. In my mind, if the whole premise is some kind of absolute trust, this should never happen.[5] And if this is regularly happening, it may be that the underlying premise is incorrect!

In 2014, a non-profit organization known as *Let's Encrypt* formed, with the idea that at least you shouldn't have to pay an extortion fee to have a TLS certificate. Their code is all open source and they generally seem to do a good job (although they are not without their own incidents) and they are good guys in this story (unless they ban me for my hijinks).

One of the ways that Let's Encrypt avoids charging you extortion fees is that it's totally automated. Instead of verifying your identity and that you own the domain, it checks via a protocol called Automatic Certificate Management Environment (ACME)[9] that you, uh, 0wn the domain. In order to get a certificate, you use a challenge-response protocol, where Let's Encrypt's ACME endpoint asks you to prove that you've already hacked the server by placing a file with specific contents of its choice on it. If you want to get a certificate with wildcards in it,[6] like `*.tom7.org`, then you have to make specific DNS records to prove that you already pwn3d the domain itself.

The many dirty secrets of TLS

I'm alluding to a bit of hypocrisy here: The way you get a certificate is itself vulnerable to the same attacks it's meant to protect against. For example, if you can successfully act as a man-in-the-middle, intercepting and modifying traffic to my website using HTTP,[7] then you can ask Let's Encrypt to issue you a certificate, succeed the challenge-response protocol, and then turn your HTTP man-in-the-middle into a 🔒 Green Padlock 4096-bit man-in-the-middle. For a very conscientious site owner there are some potential protections but if someone can man-in-the-middle your site, you are already in trouble.[8] If someone gets a rogue certificate issued for your domain and you notice, you can maybe do the challenge-response yourself to convince the CA to revoke the rogue certificate. Or perhaps the attacker will be doing the same to revoke *your* certificate.

Oops—HACKED!! Even if you revoke a certificate, it might not matter! Browsers have kinda stopped checking for certificate revocation. The issues include:

- Certificate Revocation Lists are enormous. For one Let's Encrypt certificate, the corresponding CRL is 50–100 MB as of 2026. Why are there so many revoked certificates? Who knows? Anyway, downloading 100 MB before visiting a site is considered rude; browser vendors know that you need to save that bandwidth to download 84 embedded copies of jQuery and the full-screen video advertisement.
- An alternative protocol called OCSP lets you check the certificate for a specific domain.[10] But oh yeah, like CRLs, OCSP queries happen over bare, unencrypted http connections. So an attacker who wants to see what sites you're visiting could just see what certificates you're checking on.
- Also, because some people on the browser teams (and good for them) know that you mostly want to browse the web and not see error messages, revocation support is typically "fail-open;" if you can't check for revocation because the CA is having trouble or because an attacker interfered with your connection to it, then the browser will just trust the certificate anyway.
- So we invented OCSP stapling (the web server gets the OCSP assertion itself and forwards it when

5. To be honest, I should note the possibility that I am just jealous that they probably have those keyboards that slide out of the server rack's *blinkenlights* and pop up a small amber display with a direct console connection to a root shell.

6. And you do, because when inspiration strikes you late at night to create a new stupid subdomain for a new stupid project, you don't want to have to endure any creativity-killing Trials.

7. As I understand, you do need to have a pretty robust hack in place, since Let's Encrypt will try requesting the HTTP challenge from multiple sources. So this is a bit harder than doing a man-in-the-middle on the coffee shop wifi.

8. As I understand, you can turn off certain challenge-response protocols using your DNS records, and use DNSSEC to make sure those records can't be spoofed. (By the way, if this was all handled by the domain registrar, whom I already need to trust and pay a fee to, I would be a lot less irritated!) You can also actively and continuously monitor the Certificate Transparency logs and cryptographically verify the 100+ entries added per second, looking for someone issuing an unauthorized certificate for your domain.

you connect), and a way to tell everybody that a user should not trust your site unless you staple OCSP. But since a typical OCSP response is valid for 7 days, an attacker can just keep reusing one for a week, even if the CA would have stopped validating the rogue certificate. Also, we deprecated OCSP so it doesn't really do anything any more.[11]

• So Chrome mostly checks for revocations using CRLSet, which is a compressed, "curated" subset of revocations. By "curated," they mean that it includes important sites like `google.com`, `mail.google.com`, `ad.doubleclick.net`, `wave.google.com`, `youtube.com`, and `FUNNY_SITE_HERE`. CRLSets contain about 0.35% of all revocations,[12] so as long as your website is in Lycos's "Top 5% Of The Web," you have almost a 10% chance of being included. If you are an old academic site forced to get a certificate due to ⚠ RED ALERT, your users probably won't see if the certificate is revoked.

• Firefox, to its credit, attempts to cover all revocations using Bloom filters (CRLite). There's a good chance that if a certificate is revoked, Firefox will notice within a few hours. Hours is still a pretty long time, especially since it takes only a few seconds to get a new certificate.

So, all that is to say: If you are accessing your bank's website or a major tech company's, the fact that it says "secure" probably does mean something. But if you don't know whether the site's own network is compromised, it does not mean much, other than that somebody did some chores recently (maybe a hacker on their behalf).

Other dirty secrets: Let us study the 🔒 green padlock as an illustration of "secure." This is not how you use padlocks. You must pass them through two other loops to secure them. Come on.

Did you know that when you get the required certificate, your cool new subdomain is published to Certificate Transparency logs, which is scraped in real-time by bots so that they can "security-scan" your site? Of course once your site is online you must be somewhat ready for visitors, but I think most site operators like to enjoy a quiet soft-launch of their Apache 2.0 "It worked!" home page before the scripts attack. Personally I don't mind the bots, because it means m o r e e n g a g e m e n t.

And just who are the trusted Certificate Authorities who can issue certificates for my website? I just took a gander (see `chrome://certificate-manager`) and the list contains 132 (plus another 18 from my operating system). Some of the surprises:

• There's a root certificate for my CAD mouse, which upon some investigation, seems to exist so that it can work around the `https` mixed-content restriction I complained about earlier, so that I can use my mouse in browser-based CAD tools.

• There's one from "Autoridad de Certificacion Firmaprofesional CIF." Do I trust them? It literally means Professional Firm en español, and they have 2.6 stars on Google Maps with 5 reviews, so ¿it's probably legít?

• There are several from a company called "Buypass," which certainly sounds like they just let you buy 'em to "bypass" something?

• There are several from vTrus, which maybe would be a better sight gag as ⚙ VTRUS. This is issued by iTrusChina, "one of China's first state-authorized Certificate Authority (CA) institutions." Um, *do* I trus China? When we talk about state-level actors factoring my RSA key with a billion-dollar quantum computer, like, do we think those people might also influence a state-authorized CA?

• Of course Microsoft, Amazon, Google, and GoDaddy have their own suites of CAs so they can issue certificates at will. Must be nice!

Also: Did you know that when you make an https connection, the very first message your browser sends includes the name of the host you are trying to access, in plaintext? This `ServerNameIndication` extension is discussed again below. It is true that the destination IP address itself often reveals which site you're trying to go to, but there are also many Content Delivery Networks for which this is not the case. In fact...

SSL offloading. Since setting up and maintaining `https` remains a chore that many site owners do not want, but they feel somehow obligated by ⚠ RED ALERT, there is other help! Most Content Delivery Networks offer "SSL offloading" or a "TLS termination proxy" where your server does not need to do the chore of HTTPS. For example, CloudFlare offers a free service called Flexible HTTPS that makes your site "partially secure." Cloudflare manages the TLS connection with the user, and communicates with your site over bare unencrypted HTTP (regardless of where it is hosted). It acts as a Machine-in-the-Middle that encrypts "some" of the traffic! Wow! Military Grade! The best part is that it is totally transparent to the user. The user gets the

🔒 green padlock so they can tell that they are talking to Cloudflare, and since they trust Cloudflare to not leak their data, all is well.[13] Then Cloudflare decrypts and leaks their data, as deliberately configured.

Of course nobody cares! The whole dance is to work around the fact that the site owner does not care, the user does not care, and Cloudflare wants to help them not care. I support it, but we could have saved even *more* resources by allowing regular `http` to continue to coexist. This might help attentive users distinguish between cases where their connection was encrypted and when it was not, and to know when they are talking directly to a site versus a Machine-in-the-Middle.

Arguably the "SSL offloading" approach is already getting significant straight-faced revenge on the toxic max-security people, and is barely considered aberrant behavior. So this would be a decent way for me to get my site to support `https` connections while feeling smug that I was not fully defeated. But: It requires me to do chores, to start relying on a whole additional provider, and maybe even give them my credit card number. They can enact arbitrary restrictions at any time, and take my site offline at any time.[9] Another classic approach is the "TLS reverse proxy," which does this same thing but communicates with a local HTTP server (not over the open internet). This is considered secure, since the HTTP connection happens over a private loopback, and is architecturally nice because you can separate the concerns of web serving from the concerns of security dance. Unfortunately, if I do that, then the toxic max-security people will have succeeded in forcing me to have a secure website!

Let'sn't Encrypt

And so! It is time for my petty "revenge." Every good project starts with some criteria. Here are the parameters:

I want my website to be accessible to users without them having to do anything special, like in the old days. I wasn't able to figure out a way to do this without a valid certificate (such an achievement would be a major security incident), so I'll do the chores to get a certificate.

But: As you already know from the title and tone of this paper, I resent being "forced" to make my site secure. So I'm going to do the worst job implementing TLS that I can manage. I want it to be painfully bad to security professionals, in the hopes that it causes psychological harm to some of the same people who are locking people out of my website and scaring them about their credit card numbers.

One of the things that the toxic max-security people love to tell you is that you aren't a good enough programmer to implement cryptography algorithms yourself. But these are the same cowboys who wrote OpenSSL,[14] the main source of security holes on the internet, and want you to use *that*?[10] No OpenSSL in this project. It doesn't even need to be installed.

On the other hand, I don't want people to be able to "hack" my server in the sense of gaining unauthorized access (buffer overflows, etc.). Anyone has authorization to access the public websites on there, and to observe the traffic if they want to be creepy. The site does not purport to be robust to impostors or interlopers. My goal is to provide "HTTP but worse": Roughly no security guarantees, but also slower, more complicated, and with a 🔒 green padlock.

The result is `httpu`, a TLS 1.2 reverse proxy that all major browsers will connect to, and which has been serving all of my websites for the last four months. You can try it,

`https://tom7.org`

Now I'll walk through some of its innovations. In order to explain how bad a job I did, I'll need to explain how TLS works, so this will also be a decent way to learn about the protocol.

---

9. While writing this paper I encountered firsthand yet another global Cloudflare outage. These CDNs do have impressive infrastructure, but it seems to be a mistake to assume that they will not have regular, unpredictable downtime!

10. I just looked to see if indeed OpenSSL is still an endless fount of security vulnerabilities—like, maybe I am being too hard on it?—and I saw that merely *three days ago* there were two different buffer overflow vulnerabilities (CVE-2025-11187, CVE-2025-15467) which "may also potentially enable code execution depending on platform mitigations" and ten additional "low severity" problems like incomplete validation, excessive memory allocation, and null pointer dereferences.

Secret handshake

In TLS's steady state, the client and server are sending each other data encrypted with a block cipher like AES. The tricky part of the protocol is getting into that steady state.

The initial phase of a TLS connection—the "handshake"—happens in plaintext for anyone to see. Its job is to negotiate what cryptographic algorithms will be used to encrypt the data later, and what keys to use. The server and client also decide what extensions can be used. During the handshake the server presents its certificate to the client, so that the client can check that they are talking to someone who 0wned the domain.

The classic way to establish the server's identity and secretly exchange encryption keys is RSA. The server presents its certificate, which is signed by the CA and contains the server's public RSA key. The client can verify the signature if it cares, and then send a message encrypted with the public key, which can only be deciphered using the server's private key. They use this secret channel to negotiate the encryption keys.

Too many secrets

RSA, which stands for Really Secure Algorithm,[15] is based on the believed difficulty of factoring large numbers. The server comes up with two large primes p and q, keeps them secret, and shares just $n = pq$. This is the public key.[11] To encrypt a message (a number m), the client computes

$$c = m^{65537} \bmod n$$

We say that the difficulty of RSA comes from the difficulty of factoring, but really it comes from the difficulty of computing modular roots (the 65537th root); the reverse of the above operation. It just happens that if you know the prime factors of n (as the server does), then you have an easy way to compute the order of the group (the multiplicative group of integers mod n, aka $\mathbb{Z}_n^*$), which gives you an easy way to compute the exponent d such that $x^{65537d} = x \bmod n$.[16]

Decorum requires the public key n to be large, and Let's Encrypt will only allow you to sign one that is 2048 bits, 3072 bits, or 4096 bits. This doesn't actually imply that the key is secure; for example, we can get a 4096-bit key by choosing the primes to be 7 and some random 4093-bit prime. This number would be trivial to factor, by trying small divisors, various general-purpose algorithms, or simply knowing my name. Let's Encrypt will check that you don't have *extremely* small factors like this, but it will let you have a *very* small prime divisor like 31337 (one of the most elite primes).

Oops—REVOKED!! The original version of `httpu` used a key with 31337 as one of its factors, but the certificate was revoked after about a week! Let's Encrypt lets anybody revoke a certificate for you—as a favor I guess—if they have the private key.

What happened? I don't actually know. Let's Encrypt does not notify you about a revocation (except in the sense that they publish it to the Certificate Revocation List and maybe browsers check it). The most likely thing is that someone (maybe Let's Encrypt itself) bulk-scans individual certificates after they're issued to try to find weak ones. A key with a factor of 31337 would trivially be found by any such scan that is trying at all. It's also possible that one of my "friends" who knew I was working on this did it to me out of a secretly harbored resentment.[12] Another possibility I realized is that someone may scan for keys that share factors. If one of the secret primes is *ever* reused in another key, then you can easily recover it by performing GCD on the two public keys. A plausible security mistake would be for two keys for the same domain to be created using a weak random number generator (like, "always use 31337 as the first prime"). And so a plausible security scan would be to check GCD of a pool of historic keys for a specific domain. It seems like a bit much to do $O(n^2)$ of these given that Let's Encrypt alone issues over 5 million certificates a day (Figure 1), but you can use some simple tricks to check batches in $O(n \lg(n))$ time.[18][19]

So: Someone scanning for reused primes *can* force me to have a secure website by revoking my certificate, if my keys contain constant vanity primes. This is true even for moderately big primes like

11. Technically, the public key contains the exponent e as well. Since this is public anyway, many systems require this to be exactly 65537 (one of the fastest primes); for example, Let's Encrypt will not even sign your key unless $e = 65537$. There will be other simplifications in this paper.

(hexadecimal, with a sea of 558 zeroes):

```
C0FFEE0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000D00D7
```

So we need a different `0xC00L 0xD00D` way to make the RSA keys weak. Note that while normative public keys are the product of two primes, all that is made public is n, which is composite. Although there are good ways to quickly test whether a number is prime, there is no known[13] way to test that n is the product of *two* primes. And in fact, the decryption math can be straightforwardly adapted to accommodate a public modulus with more than two prime factors. So I can actually use an RSA key of $n = p_1 \times p_2 \times p_3 \times ... \times p_k$ and nobody can quickly test that it is an unusual key. The fastest known test is to factor it.

My server uses a key with 16 prime factors. It is 4096 bits, so it is Military-Grade, but each factor is only ~256 bits, so it is Baby-Diaper–Grade—perfect for the United States Government in 2026.

How do we know what diaper size? The performance of some factoring algorithms like ECM[20] depends[14] on the size of the factor being found. So the security achieved here is more like a 512-bit modulus with two 256-bit factors.[15] This is considered very insecure; even in 2009 NIST required a minimum of 1024 bits.[21] The current world record for the largest factor found by ECM is 83 digits (276 bits),[22] so 256 bits is clearly doable.

It would still take hundreds to thousands of dollars of computer time to find a factor of my RSA modulus. If someone does, they can decrypt the traffic to my public website (and if they don't want to pay for the computer time, there will be

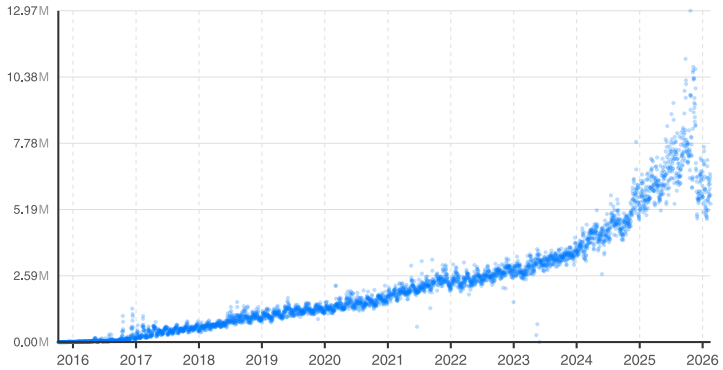


Figure 1. Daily number of certificates issued by Let's Encrypt since its birth.[17] You can see the vigorous decline in certificates that began around November 2025, which is when `httpu` first went online. (I think this is, in fact, a coincidence.)

easier ways). They can also get started revoking my keys, but here's a non-obvious twist: Although finding a single factor probably allows compromising the session keys (it reveals $m \bmod p_i$), revoking the certificate requires being able to create signatures, which needs the complete factorization. That means repeating that modestly expensive factoring 14 more times on the remaining composite. So, paradoxically, having sixteen 256-bit factors actually *increases* security compared to a single 256-bit factor and one 3840-bit factor. Since I am not trying to prevent people from snooping on the traffic, but I would rather not have them revoking the certificate, this is a favorable tradeoff.

If someone does factor the key and revokes it, then probably most browsers won't notice, and I can get another one lickety-split. I can adjust the number of bits upward if this happens too often, or even automate the difficulty setting to find the sweet spot. This is not an invitation or encouragement to spend money to revoke my key. It's more like the observation that some idiot can smash into my parked car on the street, and this sucks for both of us, but is also straightforwardly dealt with.

So that's bad RSA. Note that modern TLS prefers Elliptic Curve Diffie–Hellman Key Exchange (ECDHE) instead of sending key materials encrypted with RSA. With ECDHE, even if the RSA modulus is factored by a futuristic quantum

12. Or someone snooping on my Subversion repository! For the record it's fair game to mess with me in a fun way—as long as it's not too close to a deadline. But I would not recommend getting spoilers this way, and forbid disseminating spoilers to others!

13. Well, publicly known. Maybe the NSA knows.

14. Technically these are adult diapers.

15. And since you have eight times as many chances of finding one of the sixteen factors, I think it's really more like 509 bits.

computer, the keys can't be decoded from captured traffic.[23] Unless that same futuristic quantum computer running Shor's algorithm can also compute discrete logarithms efficiently and break Diffie–Hellman itself.[24] They call this "perfect forward secrecy," which does seem a bit optimistic given the history.[25]

ClientHell and ServerHell

On to the actual protocol. The client first sends its `ClientHello` packet, which contains the following information:

- The name of the server it wants to connect to (SNI extension).
- A 32-byte random value (client random).
- The list of ciphers and extensions it supports.

The server can look up the correct key for the server the client wants, and then respond with its `ServerHello`, containing:

- The certificate that applies, which contains the public key.
- A 32-byte random value (server random).
- The selected cipher and extensions that are mutually supported.

The goal of the cipher and extension negotiation is to make sure that the client and server agree on what options are enabled, so that new functionality can be rolled out over time and then de facto required, breaking your website. In fact, one of the ciphers offered is called `NULL` (maybe more accurately `identity`), which means no encryption. Unfortunately, this cipher is no longer supported by any browser, so we'll have to do more work to defeat the encryption. I choose the venerable (but not vulnerable) `RSA_WITH_AES_256_CBC_SHA` (`0x0035`), which is universally supported and secure if used correctly. I do not use it correctly.

The client then sends another random 48-byte string ("PMS") to the server in the `ClientKeyExchange` message; this time it is encrypted with the server's public RSA key. This is actually the only thing encrypted with RSA and the only thing so far that can't be seen by observers.

To encrypt the remainder of the session with AES-256, we need a 256-bit key. The key is chosen in a way that uses information from both the server and client: This is one purpose of the 32-bit random strings sent during the `Hello` messages. Both the server and client compute the same thing:

Hash(PMS, client random, server random)

to derive a 48-byte "master secret." They then do a similar hash of the master secret to compute the client's AES key and the server's AES key.

For the purposes of `httpu` it would be nice if we could just choose constant AES keys, like all zeroes. This would make decrypting the traffic very easy. But since the client random and PMS are inputs to the hash function (a strong hash function based on SHA-256) we don't have any way to force a specific output without the client's help. However, we can still create serious security problems. The server "random" is generally "pseudo-random," and we can make it be *very pseudo-* by always using the same value. 32 bytes is enough space to comfortably put my credit card number and expiration date in ASCII. One nice thing about this is that it elegantly makes good on the warning we saw at the beginning of the paper about attackers trying to steal my credit card![16]

Another good thing about this is that it makes the server vulnerable to replay attacks. An observer who captures the handshake knows the server and client random values, and also the encrypted PMS, which is used to derive the session key. If the server always uses the same server random, then the attacker can replay these same messages and establish the same exact keys used before. Since the attacker can't decrypt the RSA-encrypted PMS, they won't actually be able to *compute* those keys and thus won't know what they are. But the server will think everything is normal, and the attacker can send captured packets from the previous session. This can still be useful. As one obvious example, if the replayed HTTP request has an effect like posting a message to my blog, they can post that same message a second time. That's okay with me because it means more engagement.

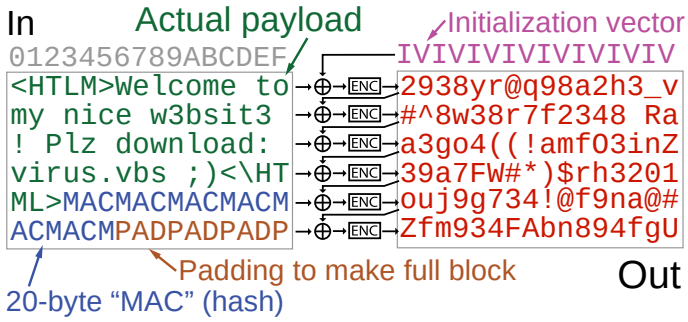
Reusing the same encryption keys can lead to other problems. For example, a very naive encryption system just encrypts each 256-bit block of the stream

16. A parallel punchline is: Rather than the normal mode of encrypting the sacred credit card numbers, we use the credit card numbers *to encrypt*.

with AES. Although the AES function can't easily be inverted, it *is* easy for an attacker to record which encrypted blocks they've seen. And sometimes it becomes easy to know something about the plaintext from these blocks. For example, you might know that the response always starts with certain bytes (because of the protocol or because it is a publicly available web page) and then you can find those encrypted bytes anywhere else in the response. Constant keys let you perform more requests, build a larger dictionary, and learn about changes in the page over time, like if the request is for `https://tom7.org/account-balance`. There are no accounts on my site and thus no balance!

This is a problem for any deterministic encryption algorithm, which is why no serious system uses this "Electronic Codebook" (ECB) mode and is instead randomized.[26] For `httpu` I chose the next worst option, "Cipher-Block Chaining" (CBC), which is still supported.

In CBC mode, each block of plaintext is XORed with the ciphertext of the previous block before being encrypted.[27] Since encrypted blocks are kind of pseudorandom, this approximates randomized encryption. So the process of encrypting a TLS record looks like this:



On the left we have the plaintext that is actually encrypted with AES, where each 16-byte line becomes a single encryption block. The green text is the actual content (e.g. the HTTP request headers or web page in response). A "MAC" is appended to the content (discussed below) and then padded out to be a multiple of 16 bytes. Each block is first XORed (⊕ symbol) with the previous block's ciphertext, using a little arrow that's so small you basically can't see it. The result of that is what's actually encrypted with AES. To kick things off, the first block is XORed with an actually-random "Initialization Vector" (IV), which is sent along with the encrypted message.

CBC has some problems, like since you know what the plaintext is being XORed with, you can flip bits in the IV and it will undetectably flip the corresponding bit in the plaintext when it is decrypted and XORed with the IV. For this reason and others, inside the encrypted message is this "Message Authentication Code" (MAC). It's a SHA-1 hash of the packet's sequence number (so that the full payload is always different in a session), some header fields, and the content. If the receiver doesn't compute the same thing after decryption, they know something's up and they're outta there! Despite being able to surgically flip bits, we can't do it in a way that keeps the MAC valid. So CBC in TLS is basically OK.

Since the server has its choice of IV, we have the most freedom on what we encrypt for the first block. In normative CBC, we do

`ENC(<HTLM>Welcome to ⊕ IVIVIVIVIVIVIVIV)`

which since IV is some random string, means we encrypt some unpredictable value like

`ENC(aA@!=3~yf93(A*C_)`

What if the IV is not random? The server chooses it for the packets it sends; it's supposed to be random but we can do whatever we want. It's a free country. If we use a fixed IV, then encryption is indeed deterministic again. A nice choice is zero, so that

`ENC(<HTLM>Welcome to ⊕ 0000000000000000)`

becomes the agreeable

`ENC(<HTLM>Welcome to)`

The block is still encrypted, but we've skipped the expensive XOR operation[28] and made it easy to reason about what's inside there. This might allow an attacker to build a dictionary of blocks, especially the first one of each packet (which uses the zero IV each time and is essentially ECB mode). That's kinda bad.

What if the server instead chooses an IV that has a particular algebraic relationship to the block being encrypted?

If we choose IV to be the same as the block being encrypted, we now have

```
ENC(<HTLM>Welcome to ⊕ <HTLM>Welcome to)
```

which is the very clean

```
ENC(0000000000000000)
```

Now we're cooking! Not only are we always encrypting the same thing (all zeroes) for the first block, but since the IV is sent along with the packet, the first 16 bytes of the packet are simply right there unencrypted in the stream! Besiegt!

This is only the first 16 bytes of the record, which is up to 2^{14} bytes in length. But the server can simply choose to send all of its data broken into 16-byte chunks, so that the first block is the only interesting block. That way, the entire stream is visible unencrypted as the IVs.

We still need to attach the 20-byte MAC and padding to form a valid record, meaning that the encrypted payload is three 16-byte blocks (all useless), plus the 16-byte IV (the data itself) and five bytes of header. This gives a nice round 69 bytes sent per 16 bytes of data, giving a data compression rate of 1:4.3. Here's an actual capture of the "encrypted" HTTP response from my site:

```
17 03 03 00 40 48 54 54 50 2f 31 2e 31 20 32 30  ‡♥♥.@HTTP/1.1 20
30 20 4f 4b 0d 12 99 ae d8 3d 8c d6 69 99 d9 e6  0 OK♪‡Ö‹╪=î╥iÖ┘µ
d2 c8 e1 b5 72 6e 29 4e c7 d5 6f 77 90 6c 01 2d  ╥╚ß╡rn)N╟╒owÉl☺-
1a ad bb 33 e0 7c c2 b6 f9 e5 24 f5 59 6d d1 69  →¡╗3α|┬╢·σ$.Ym╤i
34 a0 b4 6d 9b 17 03 03 00 40 0a 44 61 74 65 3a  4á┤m¢‡♥♥.@.Date:
20 46 72 69 2c 20 31 33 20 4d 12 99 ae d8 3d 8c   Fri, 13 M‡Ö‹╪=î
d6 69 99 d9 e6 d2 c8 e1 b5 72 e1 30 4e 12 8e 53  ╥iÖ┘µ╥╚ß╡rß0N‡ÄS
cf 8f 70 69 5b 55 fa 5f c4 87 9c bb ba 03 37 cf  ╧Åpi[U·_─çÉ╗║♥7╧
66 79 79 48 b5 62 81 bc 1f 30 17 03 03 00 40 61  fyyH╡bü╝.0‡♥♥.@a
72 20 32 30 32 36 20 31 33 3a 32 32 3a 32 32 12  r 2026 13:22:22‡
99 ae d8 3d 8c d6 69 99 d9 e6 d2 c8 e1 b5 72 f1  Ö‹╪=î╥iÖ┘µ╥╚ß╡r±
6a f8 27 82 d4 db 07 f1 4a 97 b2 8b fb bc 19 39  j°'é╘█·±Jù▒ï√╝↓9
40 9d cf c4 39 d9 41 e7 f3 ee 29 fb d6 21 b2 --  @¥╧─9┘Aτ≤ε)√╥!▒
```

There are three packets here. Each starts with the five-byte header `17 03 03 00 40`, where `0x40` is the length of the encrypted data. These 64 bytes are the plainly visible HTTP response `HTTP/1.1 200 OK\r` as the IV, followed by the encrypted zeroes `12 99 ae d8 3d...` and then the encrypted MAC and padding. The encrypted zeroes are the same each time, making the stream self-synchronizing and highly compressible.

Even better for efficiency: Since we encrypt zeroes every time, we can *precompute* that encrypted value during setup and reuse it, saving valuable resources. The MAC is different each time because it also includes the packet's serial number, but we can easily increment a counter and also compute this stream in advance as well. Because it is oblivious to the data being encrypted, this approach is also completely immune to side-channel timing and power attacks.[29] This ahead-of-time, data-independent encryption is simply not possible with normcore HTTPS implementations.

The client is probably still sending packets with random IVs, so their data is still mostly safe from eavesdropping. We could modify web browsers to use ahead-of-time encryption like the server, but if we're going to do this, we might as well just disable the warning message and let them load pages over HTTP without being scared. However, most browsers support an additional extension that allows us to ensure the client data can be decrypted as well.

Ticket to Ride

Once you start doing max-security you become preoccupied with making sure everything is secure for a billion years, and then you become embarrassed at how costly it is to run the encryption algorithms. Compared to plain HTTP, TLS adds several complete round trips to the page load (`ClientHello`, `ServerHello`, `ClientKeyExchange` and `ChangeCipherSpec`, then `HandshakeFinished`, each of which depends on the previous messages and cannot be pipelined,[17] which is a problem because most users want the internet to be fast when they are browsing web sites. The RSA operations are somewhat expensive, especially if they are done inefficiently on purpose to prevent side-channel attacks. Worse, loading a typical web page over HTTP usually means making multiple connections to the server to request its different assets (e.g. in parallel); each request has to do the whole serial handshake. All this stuff makes the browser and server people yearn for a protocol that just gives them the data.

So we have Session Tickets. If the client notes that it supports session tickets and the server agrees, then the server can send a session ticket as part of the initial handshake.[30] The session ticket is an opaque blob of data that the client can save and send when establishing a future connection to bypass most of the handshake steps[18] and get right to it.

The session tickets are part of the handshake so they

are always sent in plaintext. A typical implementation would do something like store connection information in a local cache, and give the client a random key that can be used to find the session data again. The session data needs to be enough for the client and server to agree on new AES keys, and so it must contain a shared secret (the "master secret" from the original connection). Another possibility is for the server (or cluster of servers) to have some internal encryption keys,[19] and to send the session data to the client encrypted with those keys. The client just sees a bunch of bytes, but when they hand them back to the server (or another server that knows the keys) it can recover the master secret and resume *without even storing any state*.

Elegant! Another elegant implementation, used by `httpv`, is to use the master secret itself as the session ticket, without any pesky other things like encryption. This requires no server-side storage or state, and has no overhead. It allows the client to initiate new sessions with its choice of master secret,[20] even if it has never connected before. It allows observers to straightforwardly decrypt connections in real time, like we want, or to seamlessly man-in-the-middle an innocent connection that is using session tickets. It's almost *too easy*!

Heartbleed support

`httpv` includes full support for the *Heartbleed* vulnerability.[32] This vulnerability exists by default in two years' worth of OpenSSL releases, from version 1.0.1 to 1.0.1g (2014). It allows a client to read 64kb of uninitialized server memory, which could contain anything: Public web pages, zeroes, or even OpenSSL itself! Forbes called it "a billion times worse than you thought,"[33] and it was generally a very exciting time for security professionals, so much so that Heartbleed has its own logo (Figure 2). According to Wikipedia, high profile hacks included the exposure of 900 social insurance numbers from the Canada Revenue Agency and impersonation of the CEO of the UK Parenting forum *Mumsnet*—serious business!

The TLS Heartbeat feature (RFC 6520) is basically ICMP ECHO for the TLS transport stream.[34] One side can send a `HEARTBEAT` message to the other with some payload, and that side echos it back. This can be used, for example, to store files.[35] Heartbleed happens because the `HEARTBEAT` message contains a length field, which does not necessarily concur with the length field of the containing TLS Record.[21] In the case that the heartbeat claimed it was long but it was not, OpenSSL's cowboy C code would read past the end of the client's packet and echo back whatever happened to be after it in RAM (or, I guess, segfault). You had to get pretty lucky for that RAM to contain something important. *My* cowboy C++ code reads from a configurable buffer where you can reveal anything you choose, so you can guarantee that it is passwords or your private files or whatever you like.

Limitations

Some applications with a high level of toxic max-security will refuse to connect to `httpv` servers because they are not able to negotiate a pretentious enough cipher suite. One example is `wget`, which will print this on a fresh install of Ubuntu in February 2026:

```
# wget https://tom7.org/papers/httpv.pdf
--2026-02-21 21:37:50--  https://tom7.org/papers/httpv.pdf
Resolving tom7.org (tom7.org)... 129.212.144.56
Connecting to tom7.org (tom7.org)|129.212.144.56|:443... connected.
OpenSSL: error:0A000152:SSL routines::unsafe legacy renegotiation disabled
Unable to establish SSL connection.
```

Even `curl` hangs up, which is somewhat surprising given that curling originates in Canada where people are much more polite.[22] It's hard to believe that someone would use `wget` with the expectation that their request is subject to ⚠ RED ALERT-level maximum security; this program just downloads the contents of the URL.[23] It is technically possible, though annoying, to configure cookies or ask wget to pass private information in

17. Though using constant client random and server random values as recommended in this paper *does* allow for some optimistic execution!

19. So one funny thing that can happen is: After going through all the expense to create ephemeral keys for "Perfect Forward Secrecy," the server can then just use fixed AES keys that it keeps in config files to encrypt the session ticket. That way, if you end up with read access to the config files, you can decrypt all sessions.[31] Perfect!

18. It complicates the handshake state machine a lot. I probably would have given up on implementing it if I cared about security, since it makes it much harder to understand the possible sequences during this delicate phase.

20. Since the server random is constant, it can know what the AES keys are before it even initiates a connection, opening the possibility of brute-forcing "vanity" AES keys offline.

Figure 2. The Heartbleed logo. This is mainly present in the paper to demonstrate BoVeX's new support for SVGs and floating figures.

the request, if you really want. But I'm pretty sure almost everybody is just using it to scrape public websites and to run commands like this:

```
curl -o- https://raw.githubusercontent.com/\
nvm-sh/nvm/v0.40.3/install.sh | bash
```

This is literally what people do with curl these days: They pipe that shit right into their shell without even looking at it.[24] The mysterious argument `-o-` is actually an ASCII emoticon for someone with their face down on their desk and their arms splayed out horizontally, in a pose of complete submission. If you are freebasing shell scripts but you are afraid that someone is going to spend 512 billion years with a massive quantum computer brute-forcing the domain's RSA key, well, you have a strange threat model to me.

The "issue" is actually easy to fix. Old TLS servers that support the esoteric "renegotiation" feature are vulnerable to a man-in-the-middle attack (CVE-2009-3555). A scoundrel connects to the server first, sends some encrypted data, and then uses the real client's initial handshake for a valid-looking "renegotiation," which then results in the client connecting to the server unaware that some data has already been sent on their behalf. With HTTP, this could cause someone to e.g. post their HTTP request and headers as a comment to my blog, by prefixing their request with `POST /cgi-bin/comment` or whatever. The fix is a new renegotiation extension that requires additional cryptographic inputs to the renegotiation, so it cannot be used as an initial handshake.[37] This doesn't matter for me because I don't support renegotiation at all. But OpenSSL gets mad if you don't announce support for the new extension, because you MIGHT be vulnerable to the issue.

Presumably the reason that OpenSSL takes this stance (which is stronger than the defaults in all regular-people browsers) is to try to induce *social shame* in system administrators who don't see the point in updating their already-working websites (perhaps because they have evaluated the technical merits of "perfect" forward secrecy and post-quantum key-exchange algorithms and decided that they weren't worth the costs and risks). They may not have anticipated someone like me, who is willing to deliberately expose himself to costs and risks for the sake of social vulnerability. Brené Brown describes the effect in her blockbuster TEDx talk *The Power of Vulnerability*:[38]

---

21. TLS is rife with redundant length fields in nested structures. It is a strange design to me: Since the outer record structure tells you exactly how long the payload is (this is necessary to compute the hash used for message authentication), it would have been easy to architect the protocol so that the parser always worked with a buffer of known size. This would reduce protocol overhead and the work to keep validating length fields whose values you should know. It also would have made heartbleed impossible.

22. Of course this is actually an OpenSSL issue; nothing to do with the nice people of Canada.

23. If I managed to bait any OpenSSL maintainers into reading this footnote, please also enjoy the top StackOverflow troubleshooting tips, like the user who writes "I resolved it by replacing https with http, and the issue was resolved. This solution might be helpful for some people." Or check in on the poor public servants at nasa.gov's Crustal Dynamics Data Information System who just want to help scientists download geodesy data so that they can know how much their GPS trails have drifted over time, but whose FAQ describes this (deliberate) "error" as a "known issue" whose only workaround is to reconfigure your system's OpenSSL.[36] *This* is your enemy?

24. The 16-kilobyte shell script installs a program that installs Node.js® so that you can install `npm` and YOLO tens of millions of lines more code onto your computer. You can tell the whole ordeal is expected to be done with eyes closed and was also written by an asshole, because early on the script will chide you `Error: the install instructions explicitly say to pipe the install script to `bash`; please follow them.`

> *And shame is really easily understood as the fear of disconnection: Is there something about me that, if other people know it or see it, that I won't be worthy of connection?*
>
> — Brené Brown

Indeed! When OpenSSL sees `unsafe legacy renegotiation` about my server, it deems it unworthy of connection. I DARE SAY THIS METAPHOR IS REALLY WORKING OUT!!

Let us dig deeper into psychoanalysis of my server.

Transfer protocol alignment chart

Obviously `httpu` is "lawful evil".[39]

You may have noticed the name `httpu` is similar to the protocol `https`, with an edit distance of one. This is actually because these protocols are named using a hithertofore undiscovered generalization of the Myers–Briggs Type Indicator (MBTI®) that includes additional categories and orthonormal basis vectors.

The most successful instrument in the field of psychometrics, MBTI® has four "dichotomies" that describe personality traits.[40] The dichotomies are **I**ntroversion vs. **E**xtraversion, **S**ensing vs. i**N**tuition, **T**hinking vs. **F**eeling, and **J**udging vs. **P**erceiving. You can quickly find out what personality type you are by taking an online quiz,[41] which results in a four-letter code like "INTP" for an **I**ntroverted, i**N**tuitive, **T**hinking, **P**erceiver.

Thus it is clear from its name that the protocol "HTTP" prefers to **T**hink rather than **F**eel and to **P**erceive rather than **J**udge. The question is: How to interpret the first two letters of its name? The answer is: The dichotomies in MBTI® are low-dimensional projections of much richer structures. We simply must use principled methods to add letters to the first two slots.

The first slot is **I**ntroversion vs. **E**xtraversion on the classic scale, which we generalize to add two new quadrants (Figure 3). In MBTI®, which was created mostly to study humans and not internet protocols, the distinction is between a singular (or inward) focus and a plural (or outward) one for energy. Jung's *Einstellungstypus* terms "introverted" and "extraverted" mostly describe human social activities which are fluid and unstructured (e.g. "After a long social event, do you feel drained and in need of solitude (I) or buzzed and looking for more interaction (E)?").[42] We can unfold this spectrum along a new additional dimension, which measures the subject's preference for organic, fluid, social interactions versus rigid, systematized, organizational interactions. This gives us two new quadrants, **A**utonomous and **H**ierarchical. The autonomous individual directs their energy inward like Jung's introvert. But in contrast, their focus is on systematizing their internal world—on tinkering as a meditative act. In organizations they are indifferent to (or even resistant to) authority. While the classic extravert is a social butterfly, the hierarchical person is interested in the natural order and their place (status) within it. In organizations they are deferential to the "chain of command."

For the second trait, MBTI® has **S**ensing vs. i**N**tuitive, which we similarly generalize with an additional dimension (Figure 4) This trait is about preferences for information processing. We use the same x-axis as MBTI®, describing whether the subject prefers information in atomic, discrete pieces (**S**ensing) vs. holistic, relational Gestalts (i**N**tuition). Again, we unfold this spectrum along a new additional y-axis. The new axis is the relationship between the subject and the world; the direction of flow of information. For people, they are usually collecting information from the world (websites), so that is why we only previously discovered the classic S/N quadrants. At the top of the chart we have this preference for discovering information as it emerges from the environment. At the bottom, we have a preference for imposing pre-existing information upon the environment. Here, the atomic/holistic dichotomy is between bottom-up reasoning (**F**oundational) and top-down reasoning (**T**axonomic). Someone who is highly foundational prefers to work from a collection of presupposed rules or individual axioms, whereas someone who is highly taxonomic prefers to impose their unified pre-existing worldview, organizing the world into their own recursively-structured categories.

We can begin to see how this expanded personality trait applies to protocols. Compare the FTP and HTTP protocols. Both of these share the **T**hinking and **P**erceiving traits. The File Transfer Protocol is neutral on the **I**ntroverted / **E**xtraverted / **A**utonomous/ **H**ierarchical trait (we could write "0FTP"); it is a client-server protocol (hierarchical) but typically exists as a single idle Unix daemon (autonomous); it has a strict definition (rigid) but is

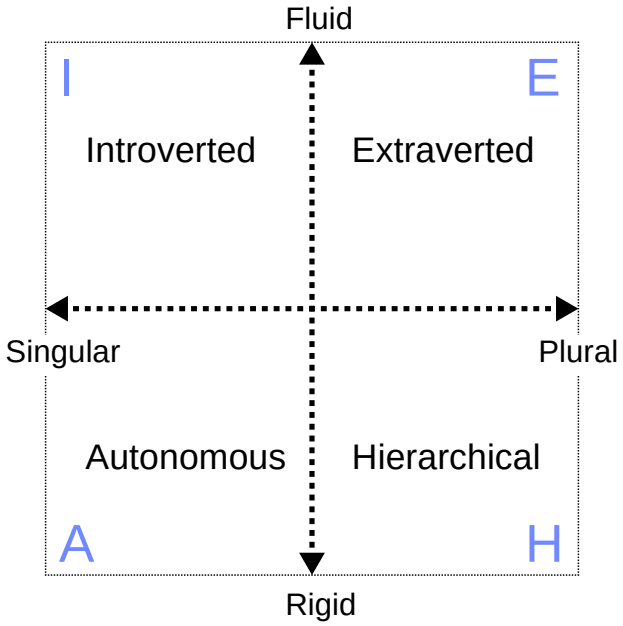


Figure 3. MBTI®'s "attitudes or orientations of energy" dichotomy generalized to a tetratomy. The classic traits are described by whether energy is directed inward (introversion) or outward (extraversion) in MBTI®. Here, we reframe the x-axis slightly to describe a singular vs. plural focus, which continues to explain the classic concepts well. A new y-axis measures whether the form of energy transfer is fluid and social (top) or systematized and organizational (bottom).

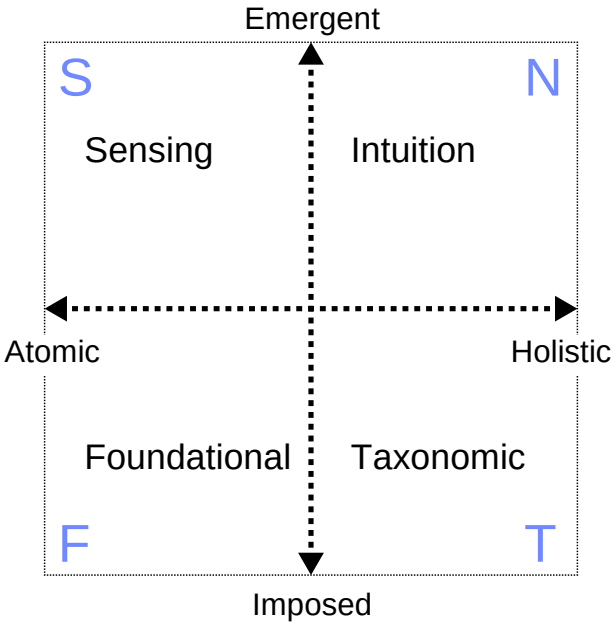


Figure 4. MBTI®'s "functions or processes of perception" dichotomy generalized to a tetratomy. The classic traits are about preferences for information processing, with the x-axis being about direct perception by the five senses[25]25. Can I put footnotes here? (sensing) versus the recognition of patterns and interrelationships (intuition). Here we generalize this x-axis to be about atomic vs. holistic information processing. A new y-axis is about whether the preference is for emergent vs. imposed structure. The classic S/N dichotomy mostly describes processing of *a posteriori* information received by the person. For a person who prefers *a priori* processing, the atomic trait is about a bottom-up or "foundational" style, as opposed to a holistic top-down or "taxonomic" style.

often used interactively for ad hoc bidirectional file transfer (fluid).[43] In contrast, HTTP is obviously **H**ierarchical. The server and client have clear roles. The protocol strictly begins with all of the headers from the client, and then returns all the data from the server; not an ad hoc interactive back-and-forth. The HTTP server typically orchestrates backends and has a pool of workers for scale. Comparing **F**TP and H**T**TP on the second trait, we see that FTP is naturally more **F**oundational, as it works at the unit of individual files (data points). HTTP is more **T**axonomic, in that it fundamentally works with tree-structured URLs. Everything works out cleanly, almost as though the personality test itself was created to fit the narrative!

Finally, and unsurprisingly, we have a wholly new trait for emotional permeability, which gives us the last letter of the personality code. On the one hand, we have a **S**ecure or guarded personality, and on the other we have a **V**ulnerable or open personality. One standard framing of secure vs. vulnerable is in attachment theory.[44] Being secure is painted as a clearly positive trait (high self-esteem; worthy of love; views others as reliable), whereas vulnerability is seen as negative (contingent self-esteem; high neuroticism; chronic anxiety; distrustful of others). Does this sound like an accurate description of HTTPS's notion of "Secure"? In my view its personality is much more similar to the negative trait described here. The https server is extremely distrustful of others, assuming that each interaction is actually mediated by an invisible adversary equipped with nation-state levels of resources dedicated to deceiving it during banal tasks. It distrusts its own code, actively scrubbing secrets from memory, and running with address-space randomization and locking itself inside a chroot "jail." You get the sense that if it were an option, it would simply detach from the world completely. To better describe the Secure/Vulnerable dichotomy we can use a more neutral framing. For example, Byrne describes a spectrum of "repressors" (guarded) vs. "sensitizers" (transparent), where a sensitizer is emotionally attuned and open at the expense of possibly experiencing more distress.[45] Similarly, Aron *et alt. eiusd. nom.* describe this same idea as "Sensory Processing Sensitivity," where the low-SPS person is tough and emotionally steady, but potentially closed off to subtle cues from their environ-

ment. The high-SPS person wears their "heart on their sleeve," easily overwhelmed, but capable of deep empathy and interpersonal openness.[46] The OpenSSL project clearly values openness (e.g. see the section on Heartbleed)! This more neutral framing is what I propose for the **S**ecure vs. **V**ulnerable dichotomy, and I think it accurately describes the personalities of HTTPS versus HTTPV.

This work in this section is impeccable—just try to pecc it and see what happens!—but to avoid the appearance of a conflict of interest, we should use at least one other independent psychometric technique to assess the personality of `httpv`.

Which NARUTO character are you? I took BuzzFeed's "Which NARUTO character are you?,"[47] answering questions on `httpv`'s behalf, but I quickly became unable to answer questions like

Q. Who did you train under?

- Jirayah
- Orochimaru
- Tsunade
- Kureni
- Asuma
- Asuma sensei

which seem to presuppose that I *am* a NARUTO character. Therefore this approach yielded no useful data.

Niceties

Most of the software I make is quite hostile to users of it. This is for a few reasons: I generally make the software for my own use, and that use is typically self-flagellative, and so if you are using the software then you are clearly asking for the flagellum (example: file system that stores your data in hundreds of thousands of live emulated Tetris games). But also, there's a selective pressure here that is—if not intentionally so, then certainly by welcome byproduct—a sort of autonomous tech support talisman. This is sort of like those robot phone trees that, when you call the bank's technical support to tell them that their TLS certificate has expired, the bots ask if you want to do a menu of somethings that can be trivially accomplished on the bank's website, with the idea that they can ward off the majority of their tech support calls without a word spoken to human ears. If the first paragraph of the documentation is "You will need a modern C++ compiler," it increases the quality of the e-mail inquiries to a manageable level.[26]

But! This time, you know, the art project is best realized if the `httpv` server is used by people other than me; maybe even *multiple* people. So, this time, I tried to make the software easy to use (at least as far as makes sense for a linux daemon). It is documented in the file `INSTALL`, which does begin by saying that you will need a modern C++ compiler.[27]

`httpv` is manually compiled and installed; the way I like it. There are simply no mysterious files. `httpv` stores all its configuration in `/etc/httpv/`. A single text file, `config.txt`, tells it what websites you have, how dumb you want the TLS malicious compliance to be, and what your credit card number is.

You can use the `multi-gen.exe` program to make yourself RSA keys. These can be 4096 bits so that it is "military-grade encryption" but also have as many factors as you want[28] so that it is almost arbitrarily insecure. You can also just use OpenSSL to generate keys, as long as they are RSA.

The accompanying program called `renew.exe` will update your certificates from Let's Encrypt if they need to be updated. You can run that in your crontab every day (or every 10 seconds, if you want) or manually whenever someone successfully navigates your phone chatbot filters to tell you that your certificate has expired. Alas, in order to get a certificate from Let's Encrypt, you need to endure the Trials of ACME. You could probably just use `certbot` to update your certificates. I wish I could offer you a way to skip certificate renewal, but this step is *the* essential inconvenience of TLS.

The `httpv` server itself can be run manually the way you like to do it, but I recommend `systemd` and supply an example config file.

---

26. It is also, not coincidentally, akin to regularly having a few minor math lectures as digressions in one's YouTube videos.

27. You can find the project site at `http://tom7.org/httpv`.

28. The theoretical limit is 4096 factors, if every factor is 2. The practical limit is more like 428 factors, since Let's Encrypt will automatically reject your key if it has a factor smaller than 757[48] (and $757^{428} < 2^{4096} < 757^{429}$).

The server itself is not super efficient (it forks on every connection, for example) but honestly just writing this in a compiled language and being thoughtful about dependencies goes a very long way compared to peers. The process uses 3MB of RAM, 2.6MB of which is shared. Once the server is running it drops root privileges, accesses no files, and allocates only a few tiny buffers. Chrome's own ⚠ RED ALERT page from the introduction uses ten times as much memory.

Even configured with the dumbest settings, it uses only about 1/3 of the CPU time of Apache in the steady state. Having looked at the source code of software in this category, I would also claim that the source code is much less difficult to understand and "audit" than its peers. For example, Google has a fork of OpenSSL (I guess it was getting out of hand), which is called "BoringSSL."[49] The git repository is 572 megabytes, and contains 445,000 lines of C/C++ code as of 2026! And this is *just* the cryptography library! Of course, BoringSSL has all sorts of support for hardware encryption, assembly-optimized routines, and robustness against timing and power attacks.[50] But I also don't care about that! I merely want people to be able to access my website and not to be able to "hack in" to my server (through buffer overflows, etc.). `httpv` is under 10,000 lines including tests and everything.

BoVeX improvements

You may be happy to know that I have continued building my own alternate universe of software tools for writing SIGBOVIK papers the hard way, with the centerpiece being the typesetting system BoVeX. I wrote it as what is seeming decreasingly like a joke for SIGBOVIK 2024.[51] In fact, during the course of this project, I spent far longer working on BoVeX itself than on the code that is the subject of this paper.

Compliance. Being in a mood for rules, whether the compliance with them malicious or not, I fixed a number of picky technical issues with BoVeX's PDF backend. These did not affect display (that I know of) but did cause some validators to barf with helpful diagnostics like `The document structure is corrupt` (Adobe Preflight) or `FAIL minimal.pdf 1b` (VeraPDF). Now I believe the generated documents are formally correct per the ISO standard.[52] They nearly comply with stricter archive standards too: This document only contains violations of clauses 6.7.3–6, 6.2.4–3, 6.7.3–4, 6.7.3–3, 6.7.3–7, 6.7.3–2, 6.4–3, 6.7.2–1, 6.4–6, 6.3.4–1, 6.2.3.3–3, 6.3.6–1, and 6.2.3.3–1 in ISO 19005 (PDF/A),[53] which is likely a world record achievement in PDF/A compliance.

Layout improvements. I made many minor improvements to correctness and typography. For example, font spacing was 0.8% too generous in the previous versions, thanks to an inherited misunderstanding about 1000 versus 72×14. I support an "infinite" penalty for overlapping text now. I fixed a bug in the core boxes-and-glue algorithm where the optimization procedure was ignorant of the penalties for expanding and contracting glue, which was why the `hfill()` utility wasn't doing what you expected. You never saw the consequences of these because I would always work around them with hacks in the run-up to a SIGBOVIK deadline, but it was probably the single most embarrassing bug. That'll teach me to just roll my own instead of copying Knuth's actual algorithm![54]

Performance. When I am writing a paper, I'm constantly compiling it to admire its many typographic problems, and to procrastinate. Papers that are longer than a few pages took irritatingly long to compile; something like 10–20 seconds on a modest cloud server. As an efficient means of procrastination, I improved the performance of every expensive phase of BoVeX's compiler, from the parser (the combinator library was rebuilding the parser objects whenever it recursed) to the frontend's simplifier (inlining was doing an $O(n)$ free variable calculation at each AST node) to its dataflow-based optimization phase (the dataflow state was using `std::unordered_set`; a bitset approach is 10× faster). Compiling a paper is two to three times faster now, and most of the time is now spent executing BoVeX code itself to run your document's expensive custom layout algorithms, as it should be.

SVG. In order that I could make the little ☑ in the opening paragraph without resorting to bitmap images (which would only allow you to print the document on finite-resolution devices), BoVeX now supports vector images via a significant subset of the SVG format.[55] This is a horrific task that I of course did entirely by hand. Like PDF, SVG is impossibly complex and has probably never been completely implemented, but I made sure to support the features that I see in Adobe Illustrator ex-

ports so that I can at least personally get it to look like I want forevermore. This includes all the path types (except arcs), basic text support, rounded rectangles, the three different ways that strokes and fills can be transparent, dashed lines, clip paths, transforms, and fill rules.

Footnotes. I can't believe I wrote two whole papers in BoVeX without any real support for Footnotes! (**footnote**: In the previous paper[56] they looked like some printer's devil misunderstood what the word footnote means, perhaps that "foot" meant "at the foot of the line" but then using PDF's upside-down coordinate system, and that "not" "e" meant **n**ot **e**nough ink. The result was horrible, like this.) Now basic footnotes are here, via a rewrite of the page layout algorithm (which is in BoVeX code; `layout-base.bovex` and `layout.bovex`). You know, one of the reasons I longed to write my own typesetting system in the first place (since the 1900s, even) is that LaTeX frequently disappoints with its irregularity. Stuff like: You can't use `\verb` in a footnote unless you discover the package `fancyvrb` (really) or if you try to put a footnote in a footnote you need to do some bizarre workaround things, or that you discover `minipage` thinking that this will solve your problems—but now you have two problems. Implementing footnotes has given me new appreciation for why it doesn't just do what you want (i.e. allow a recursive structure where footnotes or minipages allow the exact same things you can do elsewhere). The process of laying out the page involves multiple cyclic dependencies[29] and so I make a series of provisional approximations that do understand footnotes as a special kind of thing, and enjoy shortcut assumptions like "a footnote cannot be longer than an entire page." Of course as I wrote the code I'm thinking about how to make it more regular, and prepping for necessary future features like non-uniform columns. But it was hard enough to get it working at all that I can easily identify with Leslie Lamport running out of time on whatever paper deadline he was *actually* supposed to be working on, and figuring that he'll just get around to those TODOs for *next year's*.

Future work

TODO

Conclusion

The SIGBOVIK style is my favorite form of expository writing, but the whiplash between offputtingly technical and lowbrow shitpost can rightly be accused of being an irony shield, enabling me to have no real stance or point. Connoisseurs who have read more than one Tom 7 paper might even be confused about how I "love cryptography" but previously shat on Blockcoin Cryptopia[35] and now apparently also hate centralized authority? Paternalistic Certocracy is *also* no good? Does Tom just have a vendetta against blockcoins because he lost his life savings investing Hawk Tuah Coin on Polymarket futures that Donald Trump would commute Ronald McDonald's 2007 pedophilia sentence before a Bollywood remake of *Forrest Gump* was released? No. This is not the only reason. I believe there is no mere dichotomy in play. Society need not be based on "code-is-law" at all!

To make the core of my disgust reaction perfectly clear here, only the following things about TLS offend me:

- You must submit to the Certificate Authority, even if you don't care and your users don't care
- The browser acts like if you didn't do this, it's somehow "dangerous"
- These two effects in tandem can get worse at any time, and have done so

The situation is annoying in principle because of the un-democratization of the web, and annoying in practice because I gotta do all these damn chores for dubious benefit (I am NOT a bank). Passive mass surveillance is a real thing, but this can be defeated with tinkerer-friendly approaches like point-to-point encryption (trust on first use). SSH is great.

Speaking of being shushed: Does the idea of being locked out of your own website bother you? Perhaps even more so than the idea that someone

29. In order to place the lines of a paragraph nicely to avoid orphans, you need to know how the paragraph breaks into lines, which requires knowing how much space each line can use. To know that, you need to know how much space you have coming up in the document, and to do that, you need to know how many pages there are going to be, and how much space in each column will contain figures or footnotes. To do *that*, you need to know how long and wide each footnote will be, which requires knowing how much space you have for footnotes and where they are allowed to start. But to know where footnotes are allowed to start and how much space will be in each column, you need to know where the paragraphs are placed.

who can manipulate internet traffic can pretend to be your website? Or the possibility that Certificate Authorities could simply decide that enough is enough with your miscellaneous malfeasance, and refuse to issue you new certificates? (Let's Encrypt does ask you to sign some shrinkwrap agreement seemingly with the idea that they can change their minds, and CAs have been known to revoke certificates for purely political reasons.) Even if you love to be Authorized, the fact that they change requirements and protocol support regularly should worry you from a forced toil perspective.

Do users care about this at all? Surely they want it to be illegal and unusual for someone to trick them about their bank and steal their money. But it is not at all clear to me that even fairly sophisticated users check the domain names that they visit, especially when they are not doing something sensitive. This is a revealed preference: If you ask them if they care about whether they're visiting the "real tom7.org" they will probably say Yes, I don't want to visit tom7.org at all and especially not the "real" one, but when they are just browsing they do not take even the most basic actions to check. The whole premise of TLS is that users reason about who they're talking to by way of the domain name. And I'm *very* confident that they don't look at who signed the certificate and are unaware of what happens to their data after the first hop.

Meanwhile there is an alternative explanation for the forced-feeding of HTTPS Everywhere which seems cynical but plausible to me. Couldn't it be that one true motivation is to increase the administrative "legibility" of the internet for the sake of commercial (or even state) interests? Can you think of anybody who demands to see documentation that you are allowed to exist as much as the Transportation Security Administration, Immigrations and Customs Enforcement, Customs and Border Protection, Paw Patrol, etc.? Are they the good guys? Surely some of what they do is a legitimate function of government—and there are real problems out there—but it's also easy to see how the pursuit of max security can easily devolve into something toxic. One irony is that exactly concurrent with the development of TLS in the late 1990s, social scientists were documenting some of the pernicious effects of this same sort of top-down structured organization of society. James C. Scott's influential *Seeing Like a State* documents many examples of how these attempts to increase legibility by authorities have led to fragile monocultures and reduced autonomy.[57] This does not need to reflect ill intent to still give rise to a real and deleterious effect. I love the subtitle of Scott's book: "How certain schemes to improve the human condition have failed."

To me, the natural alternative to the administrative state is not the anarcho-capitalist casino that blockcoin boys have built. It's one where localized expertise and social processes are used for reasoning about all sorts of considerations, including trustworthiness. In fact, where trustworthiness isn't even a common concern, because the typical activities are mostly collaborative and abuse just isn't that fun or rewarding. More like Wikipedia than w e b 3. Of course I know that this is somewhat naive, but I also know it to be a good ideal. And I did see it thrive once during the best days of the internet.

Cryptography is not really the enemy, but it regularly shows up as a tool of computer science traitors (app store signatures, drinking rights management, locked-down computers, ransomware) because of its exceptional ability to deny access. A good technologist should be fascinated by such power, and also very thoughtful about what they set in motion with it.[58] But perhaps a new subfield ought to counterbalance the prevailing toxic maxsecurity. I can now attest that it is a subversive joy to artfully botch "security" software, especially if you imagine the faces cringing! Undoing the coat of red paint indicated by the iconic *Applied Cryptography*,[59] we could imagine a field of *Gymnography* (from the Greek *γυμνὸσ*, naked) or even the more active and downright catchy *Strippedography*. Papers, please!

Bibliography

[53] ISO/TC 171/SC 2. "ISO 19005-1:2005 Document management – Electronic document file format for long-term preservation – Part 1: Use of PDF 1.4 (PDF/A-1)". *International Organization for Standardization*. Geneva, Switzerland, October 2005. 29 pages.

[52] ISO/TC 171/SC 2. "ISO 32000-1:2008 Document management – Portable document format – Part 1: PDF 1.7". *International Organization for Standardization*. Geneva, Switzerland, 2008. 747 pages.

[17] https://letsencrypt.org/stats/. Internet Security Re-

search Group. “Let's Encrypt Stats”.

[20] Lenstra Jr., Hendrik W.. “Factoring integers with elliptic curves”. *Annals of Mathematics*, 126(3), JSTOR. November 1987. pp. 649–673.

[8] https://sslmate.com/resources/certificate_authority_failures. SSLMate. “Timeline of Certificate Authority Failures”. 2026.

[11] Josh Aas. “Ending OCSP Support in 2025”. *Let's Encrypt Blog*. December 2024.

[25] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, Paul Zimmermann. “Imperfect forward secrecy: How Diffie-Hellman fails in practice”. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. October 2015. pp. 5–17.

[3] Devdatta Akhawe, Adrienne Porter Felt. “Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness”. *22nd USENIX Security Symposium*, USENIX Association. Washington, D.C., August 2013. pp. 257–272.

[46] E. N. Aron, A. Aron. “Sensory-processing sensitivity and its relation to introversion and emotionality”. *Journal of Personality and Social Psychology*, 73(2). August 1997. pp. 345–368.

[21] Elaine Barker, Lily Chen, Andrew Regenscheid, Miles Smid. “Recommendation for pair-wise key-establishment schemes using integer factorization cryptography”. *US Department of Commerce, National Institute of Standards and Technology*, Special Publication 800-56B. August 2009. 114 pages.

[9] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, James Kasten. “Automatic Certificate Management Environment (ACME)”. *IETF*, RFC 8555. March 2019. 95 pages.

[5] Karthikeyan Bhargavan, Gaëtan Leurent. “On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN”. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. October 2016. pp. 456–467.

[7] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, David Cooper. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”. *IETF*, RFC 5280. May 2008. 151 pages.

[44] John Bowlby. “Attachment and loss – Attachment (Vol. 1)”. *Basic*. New York, 1969. 428 pages.

[38] Brené Brown. “The power of vulnerability”. *TEDxHouston, Houston, TX*. June 2012. https://www.ted.com/talks/brene_brown_the_power_of_vulnerability.

[45] Donn Byrne. “Repression-sensitization as a dimension of personality”. *Progress in experimental personality research*, 72. 1964. pp. 169–220.

[13] Kate Conger. “Major Cloudflare bug leaked sensitive data from customers' websites”. *TechCrunch*. February 2017.

[6] Joan Daemen, Vincent Rijmen. “Rijndael: The advanced encryption standard”. *Dr. Dobb's Journal*, 26(3), Miller Freeman Inc.. 2001. pp. 137–139.

[55] Erik Dahlström, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathan Watt. “Scalable Vector Graphics (SVG) 1.1 (Second Edition)”. *World Wide Web Consortium (W3C)*. August 2011.

[23] Whitfield Diffie, Van Oorschot, Paul C, Michael J Wiener. “Authentication and authenticated key exchanges”. *Designs, Codes and cryptography*, 2(2), Springer. June 1992. pp. 107–125.

[2] Cory Doctorow. “Enshittification: Why Everything Suddenly Got Worse and What To Do About It”. *Verso Books*. October 2025. 192 pages.

[33] Bob Egan. “A Billion Smartphone Users May Be Affected by the Heartbleed Security Flaw”. *Forbes*. April 2014.

[27] William F. Ehrsam, Meyer, Carl H. W., John L. Smith, Walter L. Tuchman. “Message verification and transmission error detection by block chaining”. *US Patent and Trademark Office*, US4074066A. February 1976. 17 pages.

[48] https://github.com/letsencrypt/boulder/blob/main/goodkey/good_key.go. Let's Encrypt. “good_key.go”. 2026.

[49] https://boringssl.googlesource.com/boringssl/. Google etc.. “BoringSSL”. 2026.

[29] Oded Goldreich, Rafail Ostrovsky. “Software protection and simulation on oblivious RAMs”. *Journal of the ACM (JACM)*, 43(3), ACM. New York, NY, USA, 1996. pp. 431–473.

[26] Shafi Goldwasser, Silvio Micali. “Probabilistic En-

cryption". *Journal of Computer and System Sciences*, Academic Press. 1984. pp. 270–299.

[39] Gary Gygax. "The Meaning of Law and Chaos in Dungeons and Dragons and Their Relationships to Good and Evil". *The Strategic Review*, 2(1), TSR. February 1976.

[18] Nadia Heninger, Zakir Durumeric, Eric Wustrow, J Alex Halderman. "Mining your Ps and Qs: Detection of widespread weak keys in network devices". *21st USENIX Security Symposium (USENIX Security 12)*. 2012. pp. 205–220.

[36] https://forum.earthdata.nasa.gov/viewtopic.php?t=6718. "CDDIS FAQ: File Downloads". April 2025.

[4] https://developer.apple.com/support/terms/apple-developer-program-license-agreement/. Apple Inc.. "Apple Developer Program License Agreement". Cupertino, CA, 2023.

[42] Carl Gustav Jung. "Psychologische Typen". *Rascher Verlag*. Zürich, 1921. 708 pages.

[54] Donald E. Knuth, Michael F. Plass. "Breaking paragraphs into lines". *Software: Practice and Experience*, 11(11), Wiley Online Library. November 1981. pp. 1119–1184.

[50] Paul C Kocher. "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems". Neal Koblitz, ed. *Advances in Cryptology — CRYPTO '96*, Springer-Verlag Berlin Heidelberg. Santa Barbara, California, August 1996. pp. 104–113.

[12] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, Christo Wilson. "CRLite: A scalable system for pushing all TLS revocations to all browsers". *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE. May 2017. pp. 539–556.

[28] Gottfried Wilhelm Leibniz. "Explication de l'arithmétique binaire, qui se sert des seuls caractères O et I avec des remarques sur son utilité et sur ce qu'elle donne le sens des anciennes figures chinoises de Fohy". *Mémoires de mathématique et de physique de l'Académie royale des sciences*. 1703.

[19] Arjen K Lenstra, James P Hughes, Maxime Augier, Joppe W Bos, Thorsten Kleinjung, Christophe Wachter. "Ron was wrong, Whit is right". *Cryptology ePrint Archive*. 2012.

[47] https://www.buzzfeed.com/m4shy/which-naruto-character-are-you-3alik. m4shy. "These 17 Questions Will Reveal Which "Naruto" Character You Are". August 2025.

[15] Joseph Menn. "Secret contract tied NSA and security industry pioneer". *Reuters*. San Francisco, December 2013.

[51] Tom Murphy VII. "Badness 0 (Knuth's version)". *SIGBOVIK*. April 2024. 16 pages.

[35] Tom Murphy VII. "Harder Drive: Hard drives we didn't want or need". *SIGBOVIK*, Association for Computational Heresy. April 2022. pp. 259–277.

[56] Tom Murphy VII. "Some upsetting things about shapes (that we already knew)". *SIGBOVIK*. April 2025. pp. 342–368.

[40] Isabel Briggs Myers. "MBTI manual: A guide to the development and use of the Myers-Briggs Type Indicator" (3rd edition). *CPP*. Mountain View, CA, 2003. 116 pages.

[1] Steven Pemberton, Daniel Austin, Jonny Axelsson, Tantek Çelik, Doug Dominiak, Herman Elenbaas, Beth Epperson, Masayasu Ishikawa, Shin'ichi Matsui, Shane McCarron, Ann Navarro, Subramanian Peruvemba, Rob Relyea, Sebastian Schnitzenbaumer, Peter Stark. "XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition) – A Reformulation of HTML 4 in XML 1.0". *World Wide Web Consortium (W3C)*. August 2002.

[43] Jon Postel, Joyce Reynolds. "File Transfer Protocol (FTP)". *IETF*, RFC 959. October 1985. 69 pages.

[41] Heide Priebe. "The Definition Of Hell For Each Myers-Briggs Personality". *Thought Catalog*. May 2015.

[37] Eric Rescorla, Marsh Ray, Steve Dispensa, Nasko Oskov. "Transport Layer Security (TLS) Renegotiation Indication Extension". *IETF*, RFC 5746. February 2010. 15 pages.

[16] Ronald L. Rivest, Adi Shamir, Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". *Communications of the ACM*, 21(2), ACM. New York, NY, USA, February 1978. pp. 120–126.

[30] Joseph Salowey, Hao Zhou, Pasi Eronen, Hannes Tschofenig. "Transport Layer Security (TLS) Session Resumption without Server-Side State". *IETF*, RFC 5077. January 2008. 20 pages.

[10] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, Carlisle Adams. "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP". *IETF*, RFC 6960. June 2013. 41 pages.

[59] Bruce Schneier. “Applied Cryptography Second Edition: Protocols, algorithms and source code in C”. *John Wiley & Sons*. New York, 1996.

[57] James C. Scott. “Seeing Like a State – How Certain Schemes to Improve the Human Condition Have Failed”. *Yale University Press*. March 1998. 445 pages.

[34] Robin Seggelmann, Michael Tuexen, Michael Glenn Williams. “Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension”. *IETF*, RFC 6520. February 2012.

[24] Peter W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. *SIAM review*, 41(2). 1999. pp. 303–332.

[32] Eve E. Slater, Roman W. DeSanctis. “The clinical recognition of dissecting aortic aneurysm”. *The American Journal of Medicine*, 60(5), Elsevier. May 1976. pp. 625–633.

[31] Drew Springall, Zakir Durumeric, J. Alex Halderman. “Measuring the security harm of TLS crypto shortcuts”. *Proceedings of the 2016 Internet Measurement Conference*. November 2016. pp. 33–47.

[14] John Viega, Matt Messier, Pravir Chandra. “Network Security with OpenSSL”. *O’Reilly Media*. June 2002. 386 pages.

[58] Jan Wehrheim. “Die überwachte Stadt: Sicherheit, Segregation und Ausgrenzung”. *Verlag Barbara Budrich*. Opladen, Germany, May 2012. 254 pages.

[22] https://members.loria.fr/PZimmermann/records/top50.html. Paul Zimmerman. “50 largest factors found by ECM”. August 2025.