JUST SECURITY
A FORUM ON LAW, RIGHTS, AND U.S. NATIONAL SECURITY

Feinstein-Burr: The Bill That Bans Your Browser

Julian Sanchez

April 29, 2016

Last week, I criticized the confused rhetorical framework that the Feinstein-Burr encryption backdoor proposal tries to impose on the ongoing Crypto Wars 2.0 debate. In this post, I want to try to explain why technical experts have so overwhelmingly and vehemently condemned the substance of the proposal.

The first thing to note is how extraordinarily sweeping the bill is in scope. Its mandate applies to:

> device manufacturers, software manufacturers, electronic communication services, remote communication services, providers of wire or electronic communication services, providers of remote communication services, or **any person who provides a product or method to facilitate a communication or to process or store data**. [emphasis added]

Any of these "covered entities," upon receipt of a court order, must be able to either provide the government with the unencrypted "plaintext" of any data encrypted by their product or service, or provide "technical assistance" sufficient to allow the government to retrieve that plaintext or otherwise accomplish the purpose of the court order. Penalties aren't specified, leaving judges with the implicit discretion to slap non-compliant providers and developers with contempt of court. Moreover, "distributors of software licenses" — app stores and other software repositories — are obligated to ensure that all the software they host is capable of complying with such orders.

Some types of encrypted communications services either already comply or could comply in some reasonably obvious way with these requirements. Others, not so much. Because of the incredible breadth of the proposal, it's not possible to detail in a blog post all the varied challenges such a mandate would present to diverse types of software. But let's begin by considering one type of software everyone reading this post uses daily: Your Web browser. To the best of my knowledge, every modern Web browser is non-compliant with Feinstein-Burr, and would have to be pulled from every app store in US jurisdiction if the bill were to become law. Let's explore why.

While ordinary users probably don't think of their Web browser as a piece of encryption software, odds are you rely on your browser to engage in encrypted communications nearly every day. Whenever you connect to a Web-based e-mail provider like Gmail, or log into an online banking account, or provide your credit card information to an e-commerce site, you're opening a securely encrypted HTTPS session using a global standard protocol known as Transport Layer Security or TLS (often still referred to as SSL or Secure Sockets Layer, an earlier version of the protocol). Even sites that don't traffic in obviously sensitive data are increasingly adopting HTTPS as a default (for example, your visit to this post if you're reading it at *Just Security* is HTTPS by default). Any time you see a little padlock icon next to the address bar on your browser, you are using HTTPS encryption.

This is absolutely essential for two big reasons. First, people routinely connect to the Web via WiFi access points they don't control — such as a hotel, coffee shop, or airport. Without encryption, an unscrupulous employee of one of these businesses — or a hacker, or anyone who sets up a phony "LocalCafeWiFi" hotspot to snare the unwary — could easily vacuum up people's sensitive data. Second, the Internet is a *packet-switched*network that operates very differently from traditional centralized phone networks. That means even when you're connecting to the Internet from a trusted access point, like your home or office, your data is passed like a relay race baton across perhaps dozens of different networks you don't control between your computer and the destination. (You can use a program called Traceroute to see all the intermediary points your data passes through on the way from your computer to any given website.) Without encryption, you'd have to trust this plethora of unknown intermediaries, foreign and domestic, not only to refrain from snarfing up your sensitive data, but to secure their systems against hackers looking to snarf up your data. Which, needless to say, is impossible: You'd be a fool to undertake a commercial transaction or send a private message under those circumstances. So it's no exaggeration to say that the Internet as we now know it, with the spectacular variety of services it supports, simply wouldn't be possible without the security provided by cryptographic protocols like TLS.

So how does TLS work? If you're a masochist, you can wade through the the technical guidelines published by the National Institute of Standards & Technology, but here's a somewhat oversimplified version. Bear with me — it's necessary to wade into the weeds a bit here to understand exactly why a Feinstein-Burr style mandate is so untenable.

When you open a secure HTTPS session with a Web server — which may, of course, be abroad and so beyond the jurisdiction of US courts — your Web browser authenticates the identity of the server, agrees on a specific set of cryptographic algorithms supported by both your browser and the server, and then engages in a "handshake" process to negotiate a shared set of cryptographic keys for the session. One of the most common handshake methods is a bit of mathematical sorcery called Diffie-Hellman key exchange. This allows your computer and the Web server to agree on a shared secret that even an eavesdropper monitoring the entire handshake process would be unable to determine, which is used to derive the ephemeral cryptographic session keys that encrypt the subsequent communications between the machines. (Click the link above for the gory mathematical details, or see the image on the right for a liberal-arts-major-friendly allegorical version.)

A few features of this process are worth teasing out. One is that properly configured implementations of TLS (depending on the exact algorithms agreed upon) give you a property known as "forward secrecy": Because a unique, unpredictable, and ephemeral key is generated for each session, old communications remain securely encrypted even if a server's long-term cryptographic keys — which remain the same over longer periods for purposes like verifying the server's identity — are later compromised. In economic terms, that means the "return on investment" for an attacker who manages to worm their way into a server is limited: They might be able to read the communications that occur while they remain in the system undiscovered, but they don't then get to retroactively unlock any historical communications they've previously vacuumed up. This both mitigates the the downside consequences of a successful attack and, perhaps more critically, makes it less rational for sophisticated attackers to expend extraordinary resources on compromising any given set of keys. A recent paper by a who's-who of security experts, incidentally, pointed to forward secrecy as a feature that is both increasingly important to make standard in an escalating threat environment and particularly difficult to square with a government backdoor mandate.

Some of the reasons for that become clear when we consider another rather obvious feature of how TLS functions: The developer of your browser has nothing to do with the process after the software is released. When I log into an e-commerce site using Firefox, Mozilla plays no role in the transaction. The seed numbers used to negotiate session keys for each TLS-encrypted communication are generated randomly on my computer, and the traffic between my computer and the server isn't routed through any network or system controlled by Mozilla. A user anywhere in the world with a copy of Firefox installed can use it to make secure connections without ever having anything further to do with Mozilla. And TLS isn't designed this way because of Edward Snowden or as some insidious effort to make it impossible to execute search warrants. It's designed that way because a lot of very bright people determined it was the best way to do secure communications between enormous numbers of arbitrary endpoints on a global packet-switched network.

There are any number of ways a government agency might be able to get the contents of a targeted user's TLS-encrypted HTTPS communications. The easiest is simply to demand them from the server side — but that will only work in cases where the server is subject to US jurisdiction (or that of an ally willing to pass on the data), and the server may not itself log everything the government wants. Advanced intelligence agencies may be able to mount various types of "active" attacks that involve interposing themselves into the communication in realtime, but developers are constantly striving to make this more difficult, to prevent criminal hackers from attempting the same trick — and while NSA may be able to manage this sort of thing, they're understandably reluctant to share their methods with local law enforcement agencies. In any event, those "active" attacks are no help if you're trying to decrypt intercepted HTTPS traffic after the fact.

Now an obvious, if inconvenient, question arises. Suppose a law enforcement agency comes to Mozilla or Apple or Google and says: We intercepted a bunch of traffic from a suspect who uses your browser, but it turns out a lot of it is encrypted, and we want you to decipher it for us. What happens? Well, as ought to be clear from the description above, they simply can't — nor can any other modern browser developer. They're not party to the communications, and they don't have the cryptographic keys the browser generated for any session. Which means that under Feinstein-Burr, no modern Web browsers can be hosted by an app store (or other distributor of software licenses), at least in their current forms.

How, then, should developers redesign all these Web browsers to comply with the law? The text of Feinstein-Burr doesn't say: Nerd harder, clever nerds! Love will find a way! You'll figure *something* out! But as soon as you start thinking about classes of possible "solutions," it becomes clear there are pretty huge problems with all of them.

One approach would be to make the key material generated by the browser not-so-random: Have the session keys generated by a process that looks random, but is predictable given some piece of secret information known to the developer. The problem with this ought to be obvious: The developer now has to safeguard what is effectively a skeleton key to every "secure" Internet communication carried out via their software. Because the value of such a master key would be truly staggering — effectively the sum of the value of all the interceptable information transmitted via that software — every criminal organization and intelligence agency on the planet is going to have an enormous incentive to whatever resources are needed to either steal or independently derive that information.

Another approach might be to redesign the browser so that the developer (or some other designated entity) becomes an effective intermediary to every HTTPS session, keeping a repository of billions of keys just in case law enforcement comes knocking. This would, needless to say, be massively inefficient and cumbersome: The unique flexibility and resilience of the Internet comes precisely from the fact that it doesn't depend on these sorts of centralized bottlenecks, which means my Chrome browser doesn't suddenly become useless if Google's servers go down, or become unreachable from my location for any reason. I don't have to go through Mountain View, California, just to open a secure connection between my home in DC and my bank in Georgia. And, of course, it has the same problem as the previous approach: It creates a single point of catastrophic failure. An attacker who breaches the master key repository — or is able to successfully impersonate the repository — has hit the ultimate jackpot, and will invest whatever resources are necessary to do so.

Yet another option — perhaps the simplest — is to have the browser encrypt each session key with the same public key, either the developer's or that of some government agency, and transmit it along with the communication. Then a law enforcement agency wanting to decrypt an intercepted HTTPS session goes to the developer or government entity whose public key was used to encrypt the session key, asks them to use their corresponding private key to unlock the session key, and uses that to decipher the actual communication. You can already guess the problem with this, right? That private key becomes a skeleton key that has to be kept secure against attackers, sacrificing the security advantages of forward secrecy. It has the additional problem of requiring a lot of additional anti-circumvention bells and whistles to prevent the user from trivially re-securing their session, since anyone who doesn't want the new eavesdropping "feature" just has to install a plugin or some third-party application that catches the packet with the encrypted session key before it leaves the user's machine.

There's a further wrinkle: All complex software has vulnerabilities — like the Heartbleed bug in the widely-used OpenSSL library, which made headlines last year as exposing millions of users to the risk of having their secure communications compromised and spurred a frantic global patching effort. Modern software is never really finished: New vulnerabilities are routinely discovered, need to be patched, updates must be widely disseminated and installed, and then the cycle starts all over again. That's hard enough as it is — an exercise in dancing madly on the lip of a volcano, as John Oliver memorably put it. Now there's the added problem of ensuring that a new update to fix an *unintentional*vulnerability doesn't simultaneously either break the *intentional* vulnerability introduced to provide law enforcement access, or interact unpredictably with any of the myriad approaches to guaranteeing government access in a way that creates a new vulnerability. People who don't actually have to do this for a living are awfully confident this *must* be possible if the nerds only nerd hard enough. The actual nerds mostly seem to agree that it isn't.

So far so awful. But now consider what this implies for the broader ecosystem — which doesn't just consist of huge corporations like Apple or Google, but individual coders, small startups, and the open source communities that collaboratively produce software like Firefox. In principle, a lone computer science student, or a small team contributing to an open source project, can today write their own Web browser (or any other app implementing TLS) using open code libraries like OpenSSL, and release it online. We owe much of the spectacular innovation we've seen over the past few decades to the fact that software can be produced this way: You don't even need a revenue model or a business plan or a corporate charter, just the knowledge to write code and the motivation to put it to use. Maybe the software makes you rich and launches the next Silicon Valley behemoth, or maybe the authors release it and forget about it, or move on and pass the torch to another generation of open source contributors.

If Feinstein-Burr becomes law, say goodbye to all that — at least when it comes to software that supports encryption of files or communications. Those existing open-code libraries don't support government backdoors, so the developer will have to customize the code to support their chosen government access mechanism (with the attendant risk of introducing vulnerabilities in the process) and then be prepared to secure the master key material for the effective life of the software—that or run a server farm to act as a key repository and secure *that*. As a practical matter, the (lawful) production of secure software in the United States becomes the exclusive domain of corporate entities large and rich enough to support (and at least *attempt* to secure) some kind of key-escrow and law enforcement compliance infrastructure.

The probable upshot of this proposal, then, isn't *just* that we all become less secure as big companies choose from a menu of terrible options that will enable them to comply with decryption orders — though it's important to keep pointing that out, since legislators seem somehow convinced the experts are all lying about this. It's that smaller developers and open source projects look at the legal compliance burdens associated with incorporating encryption in their software and decide it isn't worth the trouble. Implementing crypto correctly is already hard enough; add the burden of designing and maintaining a Feinstein-Burr compliance strategy and a lot of smaller developers and open source projects are going to conclude it's not worth the trouble.

In an environment of dire and growing cybersecurity threats, in other words, legislators seem determined to dissuade software developers from adopting better security practices. That would be a catastrophically bad idea, and urging developers to "nerd harder" doesn't make it any less irresponsible.

Julian Sanchez is a senior fellow at the Cato Institute and contributing editor for Reason magazine. Follow him on Twitter