Go Static or Go Home

In the end, dynamic systems are simply less secure.

Paul Vixie

Most current and historic problems in computer and network security boil down to a single observation: *letting other people control our devices is bad for us*. At another time, I'll explain what I mean by "other people" and "bad." For the purpose of this article, I'll focus entirely on what I mean by *control*. One way we lose control of our devices is to external distributed denial of service (DDoS) attacks, which fill a network with unwanted traffic, leaving no room for real ("wanted") traffic. Other forms of DDoS are similar—an attack by the Low Orbit Ion Cannon (LOIC), for example, might not totally fill up a network, but it can keep a web server so busy answering useless attack requests that the server can't answer any useful customer requests. Either way, DDoS means outsiders are controlling our devices, and that's bad for us.

Surveillance, exfiltration, and other forms of privacy loss often take the form of malicious software or hardware (so, "malware") that somehow gets into your devices, adding features like reading your address book or monitoring your keystrokes and reporting that information to outsiders. Malware providers often know more about our devices than we as users (or makers) do, especially if they have poisoned our supply chain. This means we sometimes use devices which we do not consider to be programmable, but which actually are programmable by an outsider who knows of some vulnerability or secret handshake. Surveillance and exfiltration are merely examples of a device doing things its owner doesn't know about, wouldn't like, and can't control.

Data Becomes Code

Because the Internet is a distributed system, it involves sending messages between devices such as computers and smartphones, each containing some hardware and some software. By far the most common way that malware is injected into these devices is by sending a message that is malformed in some deliberate way to exploit a bug or vulnerability in the receiving device's hardware or software, such that something we thought of as data becomes code. Most defense mechanisms in devices that can receive messages from other devices prevent the promotion of data that is expected to contain text or graphics or maybe a spreadsheet to code, meaning instructions to the device telling it how to behave (or defining its features). The failed promise of anti-virus software was that malware could be detected using pattern matching. Today we use anti-virus tools to clean up infected systems, but we know we can't count on detecting malware in time to prevent infection.

So we harden our devices, to try to keep data outside from becoming code inside. We shut off unnecessary services, we patch and update our operating systems, we use firewalls to control who can reach the services we can't shut off, we cryptographically sign and verify our code, and we randomize the placement of objects in program memory so that if data from the outside does somehow become code inside our devices, that code will guess wrong about where it landed and so fail to hurt us. We make some parts of our device memory data-only and other parts code-only, so a successful attack will put the outsider's data into a part of our device memory where code is not allowed to be executed. We log accesses to our systems, hits on our

firewalls, and flows on our networks, trying to calibrate the normal so as to highlight the abnormal. We buy subscriptions to network reputation systems so that other devices known to be infected with malware cannot reach our services. We add CAPTCHAs to customer registration systems to keep botnets from creating fake accounts with which to attack us from inside our perimeters. We put every Internet-facing service into its own virtual machine so that a successful attack will reach only a tiny subset of our enterprise.

Inviting the Trojan Horse Inside

And then, after all that spending on all that complexity for defense, some of us go on to install a DCMS (Dynamic Content Management System) as our public-facing web server. This approach is like building a mighty walled city and then inviting the Trojan horse inside, or making Achilles invulnerable to harm except for his heel. WordPress and Drupal are examples of DCMSes, among hundreds. DCMSes have a good and necessary place in website management, but that place is not on the front lines where our code is exposed to data from the outside.

The attraction of a DCMS is that nontechnical editors can make changes or additions to a website, and those changes become visible to the public or to customers almost instantly. In the early days of the World Wide Web, websites were written in raw HTML using text editors on UNIX servers, which means, in the early days of the Web, all publication involved technical users who could cope with raw HTML inside a UNIX text editor. While I personally think of those as "the good old days," I also confess that the Web was, when controlled entirely by technical users, both less interesting and less productive than it is today. DCMS is what enables the Web to fulfill the promise of the printing press—to make every person a potential publisher. Human society fails to thrive when the ability to speak to the public is restricted to the wealthy, to the powerful, or to the highly technical.

And yet, DCMS is dangerous as hell—to the operators who use it. This is because of the incredible power and elasticity of the computer languages used to program DCMS systems, and the power and elasticity of the DCMS systems themselves. DCMS gives us a chance to re-fight and often re-lose the war between data on the outside and code on the inside. Most of the computer languages used to write web applications such as DCMS systems contain a feature called **eval**, where programming instructions can be deliberately promoted from data to code at runtime. I realize that sounds insane, and it sort of is insane, but **eval** is merely another example of how *all power tools can kill*. In the right skilled hands, **eval** is a success-maker, but when it is left accessible to unskilled or malicious users, **eval** is a recipe for disaster. If you want to know how excited and pleased an attacker will be when they find a new way to get your code to **eval** their data, search the Web for "Little Johnny Tables."

But even without **eval** in the underlying computer language used to program a DCMS or in the database used to manage that program's long-term data, such as student records, most DCMSs are internally data driven, meaning that DCMS software is often built like a robot that treats the website's content as a set of instructions to follow. To attack a DCMS by getting it to promote data to code, sometimes all that's needed is to add a specially formatted blog post or even a comment on an existing blog post. To defend a DCMS against this kind of attack, what's needed is to audit every scrap of software used to program the DCMS, including the computer language interpreter; all code libraries; the operating system including its kernel, utilities, and

compilers; the web server software; and any third-party apps that have been installed alongside the DCMS. (Hint: this is ridiculous.)

DDoS

Let's rewind from remote code execution vulnerability (the promotion of outsider data into executable code) back to DDoS for a moment. Even if your DCMS is completely non-interactive, such that it never offers its users a chance to enter any input, the input data path for URLs and request environment variables has been carefully audited, and there's nothing like bash installed on the same computer as the web server, DCMS is still a "kick me" sign for DDoS attacks. This is because every DCMS page view involves running a few tiny bits of software on your web server, rather than just returning the contents of some files that were generated earlier. Executing code is quite fast on modern computers, but still far slower than returning the contents of pre-generated files. If someone is attacking a web service with LOIC or any similar tool, they will need 1,000 times fewer attackers to exhaust a DCMS than to exhaust a static or file-based service.

Astute readers will note that my personal website is a DCMS. Instead of some lame defense like "the cobbler's children go shoeless," I'll point out that the attractions of a DCMS are so obvious that even I can see them—I don't like working on raw HTML using UNIX text editors when I don't have to, and my personal web server isn't a revenue source and contains no sensitive data. I do get DDoS'd from time to time, and I have to go in periodically and delete a lot of comment spam. The total cost of ownership is pretty low, and if your enterprise website is as unimportant as my personal website, then you should feel free to run a DCMS like I do. (Hint: wearing a "kick me" sign on your enterprise web site may be bad for business.)

At work, our public-facing website is completely static. There is a CMS (Content Management System), but it's extremely technical—it requires the use of UNIX text editors, a version control utility called GIT, and knowledge of a language called Markdown. This frustrates our non-technical employees, including some members of our business team, but it means that our web server runs no code to render a web object—it just returns files that were pre-generated using the "ikiwiki" CMS. Bricolage is another example of a non-dynamic CMS but is friendlier to non-technical WYSIWYG users than something like ikiwiki. Please note that nobody is DDoS-proof, no matter what their marketing literature or their annual report may say. We all live on an Internet that lacks any kind of admission control, so most low-investment attackers can trivially take out most high-investment defenders. However, we do have a choice about whether our website wears a "kick me" sign.

There's a hybrid model which I'll call *mostly static*, where all the style sheets, graphics, menus, and other objects that don't change between views and can be shared by many viewers are pre-generated and are served as files. The web server executes no code on behalf of a viewer until that viewer has logged in, and even after that, most of the objects returned on each page view are static (from files). This is a little bit less safe than a *completely static* website, but it's a realistic compromise for many web service operators. I say "less safe," because an attacker can register some accounts within the service in order to make their later attacks more effective. Mass account creation is a common task for botnets, and so most web service operators who allow online registration try to protect their service using CAPTCHAs. (Remind me to tell you some time about CAPTCHA scraping, teenagers, and pornography.)

The mostly static model also works with CDNs (Content Distribution Networks) where the actual front end server that your viewers' web browsers are connecting to is out in the cloud somewhere, operated by experts, and massively overprovisioned to cope with all but the highest-grade DDoS attacks. To make this possible, a website has to signal that static objects such as graphics, style sheets, and JavaScript files are cacheable. This tells the CDN provider that it can distribute those files across its network and return them many times to many viewers—and in case of a DDoS, many times to many attackers. Of course, once a user logs into the site, there will be some dynamic content, which is when the CDN will pass requests to the real web server, and the DCMS will be exposed to outsider data again. This must never cease to be a cause for concern, vigilance, caution, and contingency planning.

As a hybrid almost-CDN model, a mostly static DCMS might be put behind a front end web proxy such as Squid or the *mod_proxy* feature of Apache. This won't protect your network against DDoS attacks as well as outsourcing to a CDN would do, but it can protect your DCMS's resources from exhaustion. Just note that any mostly static model (CDN or no CDN) will still fail to protect your DCMS code from exposure to outsider data. What this means for most of us in the security industry is that static is better than mostly static if the business purpose of the web service can be met using a static publication model.

So if you're serious about running a Web-based service, don't put a "kick me" sign on it. Go static, or go home.

LOVE IT, HATE IT? LET US KNOW

FEEDBACK@QUEUE.ACM.ORG

PAUL VIXIE is the CEO of Farsight Security. He previously served as president, chairman, and founder of ISC (Internet Systems Consortium); president of MAPS, PAIX, and MIBH; CTO of Abovenet/MFN; and on the board of several for-profit and nonprofit companies. He served on the ARIN (American Registry for Internet Numbers) board of trustees from 2005 to 2013 and as its chairman in 2008 and 2009. Vixie is a founding member of the ICANN RSSAC (Root Server System Advisory Committee) and ICANN SSAC (Security and Stability Advisory Committee).