Should Email Remain 'Dumb'?

Tuesday, 29 October, 2022 FOR IMMEDIATE RELEASE 'BenFranklin' Email Virus: Government agency issues urgent alert about email-script virus; cross-mailing AMP exploit spiking global email volumes 100x.

Gmail and Outlook 365 users are urged to be extremely cautious about opening messages marked "FWD:" from their most-frequent correspondents. Messages may contain the 'BenFranklin' zero-day exploit, which begins forward random inbox messages to every person in a user's address book. No activity is visible, and messages appear to be forwarded from the original user.

This is a fictional headline, set in the future.
No such "email script virus" exists.
But if email becomes scriptable, it's impossible to make it impossible.

A technical framework called AMP for Email ('Accelerated Mobile Pages'), pioneered by Google, subsequently released as open source to be stewarded by the OpenJS Foundation, is gaining some traction in the email marketplace. It allows email clients to do something they haven't done to date: run script commands locally.

AMP aims at providing useful web-like capabilities within email messages; Google's guidelines state that "*users must be able to experience the same content and complete the same actions on AMP pages as on the corresponding canonical pages, where possible.*" AMP for Email is gaining some traction at different tiers of the email service-provider spectrum, from AWeber to Adobe.

The purpose of this essay is not a security critique of AMP, or a discussion of technical standards and market power. My aim is to make the argument that introducing computing operations, however limited, and an email-specific scripting language, however limited, makes the risk of eventual security issues, lessening of user control, and erosion of trust in email inevitably higher.

Email is one of the few mainstream platforms that has stayed obstinately "dumb" — inherently incapable of computing operations of and on its own. The temptation to change that, and introduce "smart", complex functions on par with other channels and standards, is understandable. But it introduces risks that, in my opinion, cannot be avoided no matter how cleverly AMP for Email is engineered. Is the gain worth the cost?

Why Risk Is Inevitable: Fundamental Insights About Computing

Seminal computer scientist Alan Turing's paper 'On Computable Numbers' included the key insight that all computing machines are essentially interchangeable: *"It is possible to invent a single machine which can be used to compute any computable sequence."*

Turing recognized that the logical operations at the foundations of computing are universal, not specific to digital devices. Case in point: parallel-supercomputing pioneer Danny Hillis built a functional digital computer with Tinkertoys. In a brilliant leap of mathematical insight, Turing realized that the paper tape-reader of his day—and our cloud VMs—are fundamentally interchangeable. Computing machines are frequently given his shorthand description "universal machines", or sometimes "Turing machines."

Turing's insight as a major logical premise has an unavoidable corollary:

Major premise (Turing): *"All computing machines are universal computing machines, capable of running any computing operation."*
Minor premise: *"________ is a computing machine."*
Conclusion: *Therefore, any computing operation can theoretically be done on this ________ machine.*

I'll label this "Turing's Corollary" for this discussion.

Turing's Corollary implies that **the probability of a Turing machine being used for purposes other than the users' is never zero**.

Good and Bad Are Not Computational

Computing machines, like other tools, evolve and develop to perform the tasks needed of them. While in theory computers are all "universal machines", they become specialized for given purposes. The Turing machine of a game-console operating system, for example, handles graphical operations far more quickly than something like a programmable logic controller.

Purpose, however, is defined externally from computability. While this seems like a statement of basic utility — rocks make better hammers than marshmallows — in the domain of Turing machines, it also defines their fundamental vulnerability.

Turn the logic inside-out to see the structure.

Turing Machine X can perform (in theory) any computation. The purpose that the user defines for that machine, or that the maker designs it to fulfill, doesn't limit what the machine can actually do in a computational sense. It is a universal computing machine.

If someone else wants Machine X to compute something else, it's fundamentally capable of running their operation. Whether that other party has to program in a particular language or use a format specific to a given machine is beside the point. Within the logical premise of “universal computing”, qualities like “purposeful” for the user, and “non-purposeful” (or even “harmful”), are not inherent to computing itself.

The most obvious manifestation of this in the world is the vast range of programs called "viruses", "malware", and other ill names. For example, the buyer of a personal computer wants it to browse, or read email, or run a spreadsheet. Some other actor sees profit in having it report on the user, or expose their data, or run other calculations (e.g. Bitcoin). That “bad” actor may have to evade security mechanisms, and learn the particular language and structure of that machine, but at a fundamental level, the "mal" operations they desire are possible because it is a universal machine. Mal is in the eye of the beholder, not the digital innards of the machine.

Turing’s corollary in a commercial context implies that the probability of a Turing machine being used to maximize other parties’ gain is not zero. This, too, is natural; if the capability to do something more-profitable is there, and if programming these profitable actions is possible (and, critically, largely invisible), someone will probably do it. Over time, development and evolution of devices involved in commercial activities is driven by improving these invisible commercial functions, not just their user-purposed functions.

Javascript Evolution Is An Example

The Javascript Turing machine that operates within modern web browsers is a living example of this behavior. Initially derided as a toy language, with painfully slow “virtual machines”, Javascript and the VMs that run it have been refined and improved to such a degree that they are now routinely used for large and complex cloud functions, and frequently run the 'full stack' of computing tasks.

As these capabilities have advanced, so have the range of actions and “mis-actions.” On balance, the Javascript VM within web browsers is used as much (or more) by other parties than by the nominal owner/user of the browser. For example, a web page may appear “static” to the user, but it will run on average 9 Javascript “trackers” and send/receive 33 tracking requests, manipulating or aggregating data for their distant makers' commercial benefit. On a heavily-commercialized web page, Javascript tracker and script operations may run into the hundreds.

That Javascript Turing machine, in other words, is running operations that the user might well not want. But because it is a universal machine, the definition of “acceptable operation” (and control of data) isn’t really in the user’s control. That universal machine can be quietly working in the interests of distant masters.

This pattern writ large creates a constant game of whack-a-Turing, played out on the computational landscape. Browser VMs beget ads; well-meaning browser makers add ad-blockers; ad-blockers are co-opted to be ad-ware of some other sort, and so on. Ditto virus/anti, ditto apps and purchases, games and loot boxes, cyber-currency farms and clouds, etc. The cycle won’t stop precisely because purpose and permission are external to computation; any computing machine is theoretically capable of operations for which it wasn't designed or which aren't desired by the nominal owners or users.

When you allow “some operations”, you lose full control of “which operations.”

It’s Turings All The Way Down

A centrifuge spins liquid samples at high speed, separating substances of greater and lesser density; would that email clients could. Modern centrifuges run by programmable logic controllers (PLCs) can perform extraordinarily sophisticated separation. But PLCs are Turing machines, controlled by Turing machines running on/within other Turing machines. A famous, incredibly complex ‘malware’ called Stuxnet, first uncovered in 2010, attacked centrifuge PLCs.

Stuxnet exploits multiple “zero-day” (unknown) vulnerabilities in the Windows OS, Siemens Step7 software, and centrifuge PLCs to collect information and ultimately instruct the centrifuges to tear themselves apart. The Stuxnet “worm” may be one of the most elegant and complex set of “mal” computables ever designed. Some 200,000 computers have told 1,000 centrifuges to spin to death to date.

Interesting but irrelevant? Hardly. Stuxnet — singularly brilliant — exploits the stack of Turing-machine-on-Turing-machines.

Because Turing machines are logical constructs rather than physical constructs, it was inevitable that people would create machines-within-machines. This Babushka-doll architecture has been extraordinarily useful; among other things, it midwifed the Internet when enterprising grad students realized that separate computers used as “network controllers” could enable disparate kinds of machines to work on a common network. These eventually became the “network card” — yet another specialized Turing machine. Cloud computing systems like AWS and Azure are the world’s largest Babushka nest of Turing machines, with Kubernetes orchestras, VM ensembles and OS soloists playing code in a myriad of languages.

It’s an extraordinary achievement, but at a cost. Every interface between layers exposes additional vulnerabilities for operations that may not be desired. Each virtual computing machine is, almost by definition, only as secure as the others upon which it runs or rests. Questions of “good” and “bad” operations become more nebulous with each layer.

“But We Can Stop This With Good Security”

Of course, this problem of desired/undesired operations has been recognized and addressed, over and over. Super-smart teams go to extraordinary lengths to design and maintain the security systems of computing machines.

Inevitably, someone finds a way “around”, in, or through those mechanisms. It shouldn’t be a surprise, for several reasons. One, it’s logically unavoidable by virtue of Turing’s corollary. Two, the stack of logic-upon-logic-upon-logic in digital devices — particular nested machines — is too complex. It’s impossible to close every digital door, window and keyhole. Three, many (most) cyber-security mechanisms are themselves additional compute operations, externally defined as “purposeful.”

The fourth and most common vulnerability — the behavior of the people using the machines — sits right at that strange boundary condition of “purposeful operation.” If I can convince you to change your password and tell me the new one, that “security mechanism” is now a “mal” operation.

As any security expert will tell you, the only secure computer is one that is turned off; in effect, no longer operable as a Turing machine.

What Does All This Theory Have To Do With Email?

I would argue that in considering whether email should finally become “a Turing machine”, it has a great deal to with email.

The current set of Internet protocols and standards that define email do not include a Turing machine. Email clients themselves don’t run Javascript, or Java, or any other language. The HTML and CSS standards supported by email are limited, and likely don’t reach the threshold for “Turing completeness.”[1]   Email is, for lack of a better

[1] https://en.wikipedia.org/wiki/Turing_completeness

word, "dumb". While email clients run on Turing machines, up until now they haven’t contained a Turing machine of their own.

AMP proposes to change that, with the best of intentions.

“The road to hell is paved with good intentions.”
—aphorism, author unknown

AMP for Email *“allows senders to include AMP components inside rich engaging emails, making modern app functionality available within email. This dynamic email format provides a subset of AMPHTML components for use in email messages, that allows recipients of AMP emails to interact dynamically with content directly in the message.”*

These are the kind of goals with which the cycle of Yet Another Turing Machine always start. They make tempting sense.

“More than 270 billion emails are sent every day, it is the pillar of many consumer and enterprise workflows. However the content that is sent in an email message is still limited – messages are static, can become out of date, and are not actionable without opening a browser. AMP email seeks to enhance and modernize the email experience through added support for dynamic content and interactivity while keeping users safe.”

Yes, it would be great if email were functional and programmable and rich.

Yes, the limitations of email in its current form are severe compared with most other common tools and channels.

Yes, mobile use would improve with tight, constrained, super-secure script options.

“*The idea of an Accelerated Mobile Page is that the internet should be user-first, and all other content that the user doesn’t care about comes second.*” says one of the early ESP AMP adopters.   User-first may be the goal, but the machine moving us to that goal is...a Turing machine. Turing’s corollary says that user-first does not mean user-only.

The AMP team acknowledges the risk:

“If you have worked with emails before, the idea of placing a script into an email may set off alarm bells in your head! Rest assured, email providers who support AMP emails enforce fierce security checks that only allow

vetted AMP scripts to run in their clients. This enables dynamic and interactive features to run directly in the recipients mailboxes with no security vulnerabilities! Read more about the required markup for AMP Emails here."

No doubt the security designs and practices will be the best possible. But fierce security checks are like fierce security dogs; someone with a different agenda will, inevitably, bring a T-bone or tranquilizer gun, and your inbox will become the latest arena for a rousing game of whack-a-mal.

By accepting a Turing machine (scripting) in email, the probability of email being used for purposes other than the user's email will move above zero. That's not a critique of AMP, or of the security design. It's just an inevitable, logical consequence that has proved out, historically, on a staggering range of digital devices - many of them (arguably) far less complex than the Javascript VMs that will run AMP operations.

The intent may be to restrict the language to 'vetted AMP scripts.' But these scripts operate and execute within a nested set of virtual and real machines. The design and security of those machines is inherently part of the operation within the AMP machine, and their vulnerabilities are AMPs vulnerabilities.

In truth, I had been thinking about this issue in the abstract, and had already drafted the majority of this essay when direct evidence of the Turing issue showed up.

On Nov 18, 2019, Google announced that they had resolved a security flaw in AMP4Email. (https://www.zdnet.com/article/google-patches-awesome-xss-vulnerability-in-gmail/) A security researcher uncovered the potential for "DOM Clobbering" via AMP4Email. In a nutshell, a researcher realized that the HTML document object model (DOM), and the Javascript VM — the "playing field" for those limited AMP commands, if you will — could be compromised, opening the door for "mal" operations with AMP. That one example embodies some of the key issues discussed here — virtual machines, layers, "mal" behavior, and so on. This one was caught; great. It would be irrational to assume that there won't be other such issues.

From 10,000 Points of Vulnerability To 1

Every email designer knows the challenge of platform variations — the permutations of operating system, client and/or browser, version, bandwidth, CSS, HTML versions and so on are near-infinite.

Coincidentally, that's the challenge that a would-be engineer of virus, malware, spyware or adware faces today if they want to use email as the delivery channel. Because email is 'dumb' and cannot execute instructions, attacks and exploits must currently aim at the underlying platforms. A virus aimed at Windows — "download and run this .exe" — won't attack Android or OS X. A Javascript exploit can't currently do anything in the email client itself; it has to escape the dumb 'box' of email — usually by tricking the user — to do something.

In other words, pre-AMP, 'mal' operations are inhibited by the sheer breadth of platforms — because the email client itself is not the target. Giving "the email client" the capability to run computing instructions introduces a single point for attack: AMP.

The AMP response is that by limiting operations to "vetted AMP scripts", risk can be mitigated.

History suggests otherwise.

There are already email clients with Turing machine capabilities that use proprietary ("not Internet standards based") scripting languages. They illustrate the risks that AMP will face quite well.

In the heyday of Windows and Office, Microsoft attempted to unify its application suite with a common language -- VB Script. VB Script ran on Word, Excel <u>and</u> the Outlook email client. VB Script is a relatively simple language, and perhaps security wasn't top of mind in the design.

Inevitably, of course, bad actors found exploits and back doors. Email inboxes and folders are too valuable a target; if they can be targeted, they will be. So-called "macros" (programs) in VBA have done user-undesirable things in Word, Excel and Outlook. As a result, even today, Outlook is saddled with the security "cost" that goes with the VB Script "benefit." This isn't a critique of Microsoft and VBA; these "macro viruses" are just a convenient real-world example. My argument is that they were inevitable.

AMP aims to limit risk by limiting permitted scripting operations. Limiting risk and eliminating risk aren't the same result. The issue is not solely whether the limited operations in and of themselves are risky or not; how these operations are implemented 'on top of' other Turing-machine layers is material as well. What a given script command is supposed to do (and not do) is ambiguous; how that command is implemented on a given 'stack' of computing environments (e.g. operating system, email client, Javascript VM) will vary. Variance is vulnerability.

A Brief Counter-Argument: Is AMP Too Limiting?

The historical record also suggests a different issue, somewhat at odds with the main point of this essay but worth noting. Limiting operations — artificially constraining the 'compute vocabulary' — has a pretty poor track record as to security AND functionality. A relatively recent example illustrates this point.

Circa 2009, Adobe Flash was the go-to "sandbox" for rich functions and media in web browsers. It was even, albeit accidentally, the default web video format for at least a short time. Flash had its own scripting language, with a limited range of functions. Yet despite the limited operations and "special scripting language" sandbox, Flash created constant vulnerability headaches for browsers, computers and users.

Steve Jobs signed the end-of-life warrant for Flash when the iPad was released without Flash support. In his 2010 essay "Thought On Flash", Jobs explained his reasoning. In brief: Flash was proprietary rather than open; unreliable, insecure, performed poorly and drained battery, and was not well-designed for touch screens. His last and most important reason: "We know from painful experience that letting a third party layer of software come between the platform and the developer ultimately results in sub-standard apps and hinders the enhancement and progress of the platform."

There's the twist. In accepting a limited platform, with at least some whiff of "proprietary", is AMP for Email getting the worst of both ends — the vulnerabilities of a Turing machine, and the shortcomings of a third party layer of software? Will mobile pages need to be "accelerated" when 5G becomes widespread; in other words, is the definition of 'modern app functionality' the right target?

Conversely, what is the reason not to go "whole hog", and allow a full-blown Web browser with Javascript support within email messages. Or, alternatively, why not a Turing-complete, email-specific language and environment? If the answer is "it would be too insecure", that just doesn't hold up, at least in the framework of this argument. When you allow "some operations", you lose full control of "which operations."

What’s Really At Issue: Trust, Control and Manipulation

Everyday language says something about the place we psychologically accord to different digital channels. “It’s in my email” — commonplace. They are “my” messages, at “my” address, under “my control.” “Email me” is used daily; “website me” is not, nor is “app me.” What does that say?

I read my email; I view your web site. Your web page may be “personalized” for me, but you control it. Users of websites understand the implicit bargain; if you are bothering to change your web site for me, there’s a reason for it. If the page is different tomorrow, that’s your decision.

My email is “mine”, though. The address is my digital “home address.” I have both cyber and legal control, generally speaking, over who can send messages to that address. If I sign up for your service or site, my email address is likely to be used as the reliable, permanent channel for my digital identity. Password requests to “my email” are considered safe and acceptable. Messages from my email address are messages “from me.” The me/my/personal associations with this particular digital channel could go on and on.

I’d summarize the email frame as “we assume that it’s in our control.” Email has earned — rightly — a surprisingly high degree of trust. We treat email messages as solid, reliable, document-like things, despite their digital nature — legally, practically, personally and otherwise.

When a thing becomes ‘programmable’, and some other party somewhere else on the Internet has some control over the programming of that thing, ownership and control change in a fundamental way. What that thing might “do” is no longer entirely under the control of the user. The changes in behavior might or might not be visible, but other parties (or, more likely, algorithms) now have a say in what the thing does.

Programmed behavior that’s visible and explicitly at odds with the user’s aims is one issue. Popup ads in a web browser (for example) are clearly not what any user wants, but because of the scriptability (Turing-ness) of that browser, it’s difficult for them to prevent it. The adoption of “safe” browsers (Firefox, Brave) to prevent such shenanigans is an understandable response. The invisible actions and manipulations that come with programmability are another issue, and probably the more relevant set for the email community to consider.

“Smart” Connected Things Alter Trust

Making a thing both programmable and connected causes a fundamental change in control and perception. The behaviors and actions of that thing stop being localized and predictable; it starts gaining “agency” — the capability for independent or other-party-controlled action — in how we relate to it.

For example — I never had trust issues with thermostats in the past. The mercury thermostat on the wall did something quasi-smart all on its own, regulating heat against temperature. Its actions and responses were localized and linear; at temperature X, stop heating; below X, start heating. It’s a complex function, but limited enough and understandable enough that trust wasn’t really an issue.

Remotely-programmable thermostats like the Nest changed this. While improving thermostat “smarts” — programmability and remote control — improved results, the user-to-device relationship changed — probably for the worse. Trust (or lack of) is now a legitimate commercial and marketing issue for thermostats and other ‘smart’ devices. Because they are programmable and connected, the cultural Gestalt — surprisingly insightful — is that they are no longer entirely under our control.

There’s no broad question about the gains that “smart and connected” can offer. In the narrow domain of email, though, I think trading decades of perceived trust and user control for a relatively short list of new functions shouldn’t be taken lightly.

> “This thing, what is it in itself, in its own constitution? What is its substance and material?” —Marcus Aurelius

The Motive Question: Why Do This?

If you step back and look at it, the business motive to adopt AMP for email is a little questionable. Email is generally the most effective digital marketing channel (“38x ROI!”), preferred by the people using it, with the highest marketing returns. It’s routinely dismissed overlooked, but continues to deliver as other channels — more “interactive”, more “functional” — come and go.

It’s worth asking why email continues to be so effective. Is it because of habit? Trust? User control? There’s no definitive answer, but somehow the proposition that email <u>needs</u> actionable, scriptable content with ‘modern app functionality’ implies that email will become more effective with these capabilities. More functions = better results is almost a truism in the tech world, but that doesn’t mean that it’s actually true. If interactivity and functionality are indeed “better”, why is the ROI from most interactive, functional digital channels lower than that of email — at least, marketing ROI.

Does email need to evolve? Certainly. Is this the best evolutionary step? One could argue that if “software is eating the world”, as Marc Andreesen said, that it’s high time email became a software platform. I’ve stated some broad reservations, but absent an alternative — especially an industry-driven, standards-based alternative — AMP seems to be winning ‘best step available’ momentum.

I get the proposition of app-like functionality within email; app-like functions would be useful. Several things about that strike me, though.

One, it’s an absurdly low bar to set — in effect, to say “oh, goodie, now email can do the stuff that web pages were doing 10 years ago.” Two, it is — even in name — retrograde; email adopting “accelerated mobile pages” just as mobile is poised to jump to 5G bandwidth, obviating the need for acceleration-via-constrained-operations. Three, the technology method — adopting a limited set of functions on top of a language designed (originally in haste) for a different medium — smacks of a cop-out. Fourth, who set “app-like functionality” as the objective? Are users so weary of that one extra click to launch the browser that they’re lining up asking for app-like functionality, or are we buying into the “more functions = better results” mantra without asking whether it is app-like functions (or the lack of them) that prevent email from being effective or useful? Are we thoughtfully accounting for the costs, losses and potential changes to email that come with adding this relatively limited functionality?

I don’t have a definitive answer. Neither do you; neither does Google. Nobody’s done this before. It behooves the people in a place to be thoughtful about an industry to do just that.

Matthew Dunn
Founder, Campaign-Genius
January 2020
matthew@campaign-genius.com