Delivering Javascript to World+Dog

Kyle Randolph kyle.randolph@gmail.com June 2017

Background

Many SaaS startups build solutions for the enterprise and choose delivering JavaScript for its agility and delivering new customer value and ubiquity with browser runtimes. Their pitch can be summarized as, *just install this one line of JavaScript on the site and accomplish your goals*.

The background shared here is from building a security program from scratch at Optimizely, whose primary pitch is *just install this one line of JavaScript on the site and accomplish your goals via experimentation* (i.e., A/B Testing).

So engineers see the functional spec *Add "<script src=..." to site. Deliver value.*

Security people think what could go wrong? ☓

The Problem with Javascript

You can do anything with javascript. So can an attacker. That's the problem with javascript.

You're a successful SaaS startup delivering your disruptive javascript awesomeness all over the web, many of the Alexa top 1000 are using your product. News sites like CNN, e-commerce sites like hm.com, the whole 9 yards. Watch the bottom of your browser and see 'Loading from cdn.optimizely.com...' everywhere as you browse the web.

For business folks, great success!

For attackers, great opportunity!

Let's say I want to spread my anarchist political message. I probably want to get on all the top media sites across the world since that would be huge exposure organization for my cause. I could go try compromise those media websites one by one by one or I can look into the common denominator amongst many of them and see what's the common technology that they're using. At least one of these is likely third-party JavaScript value ads. So hey, lazy^H^H^H^Heconomical attacker in the house, why don't I compromise one of these third party javascripts and then I can compromise all this media websites. Pwn once, pwn

everywhere! Works for one-shot political messages, or if you can stay stealthy malware delivery, botnets, and spying.

Real World Example

A real world example of this is Gigya. Gigya has that same functional spec as Optimizely, where you load their third party javascript on your page to get some business value. In 2014, Syrian Electronic Army recognized that Gigya was loaded on many popular Western media sites, and by compromising Gigya, they could pwn all of those Gigya-loading sites. Forbes, The Independent, The Telegraph, and others all suffered. Specifically, SEA got control of gigya.com via Gigya's domain name registrar, and pointed Gigya's CDN to SEA servers. Pretty clever up until this point, you have a bunch of prominent sites lined up to do your bidding. Then the followthrough comes…. A popup that says *You've been hacked by the Syrian Electronic Army*. They could have put the payload of their choosing there, or a manifesto… but they chose a popup <shrug>. The next attacker may not be so kind as to stop at a political message.

Weakest Link: Humans

As we all know, humans tend to be the weakest link in security. The two groups of these creatures who could torpedo your saas are your engineers and the folks using your product.

Customers

First, it's not the failure on the customer's part if the platform that we provide to them doesn't give them the security they expect. Ideally we take their choices out of the equation so they don't have to choose to be secure (spoiler alert: 99% won't) but instead they're secure by default.

Developers

Startup developers want to move fast, unencumbered by big company bureaucracy. They ship code daily. More access means less headaches getting their job done.

Least Privilege

So we end up with all developers wanting admin. One word for you: sudo. Sometimes, maybe a lot of times developers need admin to do their job, but not all of the time. Try giving them a role that is read only or at least restricted by default, and an easy way to sudo to admin when needed. Log and monitor heavily. You'll get valuable data showing how much less frequently developers need admin than when they think they need it.

Managing Secrets

A sound bet on the #1 way a saas startup dies by pwnage is due to poor management of secrets. Your infrastructure likely has more than one service, and these services have to

communicate with one another. All it takes is one developer to accidentally push an admin API token to github and within a few hours the organization is hosed. We've seen variations of this nuke more than a few cryptocurrency startups. Top priority- Isolate these secrets so only prod services see them and not humans. There's great tools for this like Confidant, Keywhiz, and Credstash. Human secrets, like the admin password for your root AWS account, should go in a team password manager. Use MFA of course, with hardware tokens locked up in a secure area.

Enterprise SaaS Authentication

A company's SaaS awesomesauce is good for nothing if the authentication is bunk. Enterprise saas platform authentication Mr. have two factor authentication Lisa single sign-on two factor authentication you could are you becomes the customers responsibility in their identity provider once they're on single sign on this is true are you fine most customers won't bother single sign-on unless they have some corporate policy four-s if your customers using single-sided are you doing it right because that means paychecks are big that means and voices are big enough Torrent security scrutiny of your deal zero customers they're single sign-on even if you don't if there's passwords the last thing you want one thing do not one is your customers tire integrity of their site and your product blowing all the hell blessing was the only thing sitting between you in

A company's SaaS awesomesauce is good for nothing if the authentication is bunk. Do you want the future of your business to all hinge on one weak password chosen by one marketing noob who's interning at your biggest customer? They probably have admin on your site, and they've probably used that same variation of a pizza123 password on twenty other sites, one of which that is likely to be pwned and have its plaintext password database dropped on pastebin.

There is some good news. Enterprise SaaS is not a consumer site, and not a social network. You don't have the same business drivers for viral user growth so a lot of signups-at-all-costs best practices can be questioned and relaxed.

First off, the password. You can crank the password complexity requirements to 11. As the recent NIST 800-63-3 shows, dictate a long password length to your customers; the rest is on you to minimize password compromise by reducing brute forcing with rate limiting, banning common passwords, hashing properly, etc. For those customers who absolutely must have special characters and numbers, or passwords that expire frequently, isolate that nonsense in off-by-default security settings and leave it to them to torture themselves with compliance-mandated password best practices.

Next, two-factor auth. If you give customers this option, you may get .1% adoption of it. If you give them the ability to mandate for their organization, you may bump that up to 1%. The other 99% are your weakest link! So to 2FA the 99%ers, so that their mistakes don't ruin your business, you need opportunistic 2FA. Collect some extra authenticating info at signup, or block off some parts of your product, until they give you this info in return. Coinbase forces you to enroll in 2FA when you sign up; the burden is worth the risk mitigation. In case you didn't get the

memo, no SMS 2FA, only TOTP or FIDO for Yubikeys. Absolutely do not try to roll your own crypto 2fa like Twitter did and ultimately rolled back because it locked so many users out. Your 2FA implementation is only as good as the account recovery infrastructure that you back it with.

Even better than 2FA is single sign on. Just drop a SAML service provider in and you're set for compatibility with most organizations because they use Okta, Ping, Onelogin, etc. Then they can decide to 2FA their SSO.

We're experimenting with login sketchiness detection, where sign-ins or privileged actions with risky attributes like different user agent or different geography raise a yellow or red flag. Too soon to decide if it's worth it.

Safely Creating Business Value

The worst thing you can do is give the customer the ability to shoot themselves in the foot, namely giving them the ability to add arbitrary javascript via your service. That's giving attackers way more opportunity to add malicious javascript via your service. As much as possible, aim to only let the customer put in data or use pre-built code components.

Building and Staging

So this is the internal guts of where you translate customer's business logic into the JavaScript and data payload delivered down to the customer site.The most important thing you can do here is absolutely minimize the attack surface of this build infrastructure and the storage you stage assets in before the CDN picks them up. They're all vanilla best practices you've heard before - harden the servers, better yet go serverless, encrypt in transit, minimize access, mutual service auth and log and monitor end to end.

Finally here is where your assets prepare to get delivered to the customer site. You need to make sure that not only your code is secure but all the third party code that you're putting in. Most of your code is probably code your company didn't write, but pulled in from somewhere else, hopefully legit projects and not just Stack Overflow. Some great tools just recently have come on the market to do this expensive scrutiny, like SourceClear and Snyk. They look at all the open source components that you include, and all the components those libraries include, when building, and find any vulnerabilities. Some will even analyze to see if you use specific lines of code where the vulnerability is, others can patch automatically. Both make open source security manageable.

CDN

SaaS javascript is served from a CDN for performance reasons. CDNs have thousands of pops everywhere on the planet so it's always a very short distance to communicate with customers' visitors' devices.

Like the other parts of your infrastructure minimize the number of admins on your CDN and especially since this is where shiz could really go wrong this is where you need that single sign-on, two factor auth, privilege minimize, logging, change management etc. This is the place that you actually want that stuff going on before anywhere else because this is the point where your javascript script integrity can have one of the big targets painted on it.

DNS

This is a big deal as we all know DNS is insecure. There's ways wreck up a large part of the Internet if you compromise DNS. It can be as simple as offering free wireless in a public space, a coffee shop, and throw up some nasty DNS. Like Gigya, your domain name registrar could be hacked. Your network could be MITM'ed. It's another one of those places you got a big target on your back. And don't forget about Availability- your DNS server goes down like Dyn got DDoSed last October, their DNS was unavailable and it took a while for much of the internet on the east coast to recover. Best thing to do here is run some mock incidents before you get tested in the real world.

TLS

Once you have your infrastructure secrets and customer authentication solved, secure communication is key. DNS and other MITM attacks can compromise the delivery of your javascript. So slap TLS on it. It's not that simple.To do TLS properly, it's all in the details.

Colliding with Performance

The place where you'll encounter pushback in CDN land is performance. There's a lot of folks out there who've never seen istlsfastyet.com and presume TLS == slow. Milliseconds count with customers who closely monitor conversions, engagement, or online purchases. Two hundred milliseconds of latency may hit their bottom line.
It's mostly fast, your p95 latency should be fine with the right optimization. Where you'll still feel latency is on mobile, outside the US. TLS handshakes add multiple round trips, and the network latency ends up being the source of the lag, not the crypto, not even the asymmetric crypto in the handshake.

On By Default

Almost no one will opt out of TLS, especially if it's presented to them in the code they copy/paste to their site. If possible, enable HSTS to force TLS on your main CDN and have a cdn-tls-optional.yourstartup.com for those 1%ers who explicitly don't want HTTPS.

HTTP2

A significant benefit is HTTP2. In browsers, you only get HTTP2 if it's encapsulated in TLS. Smart folks realized you don't have to concern yourself with compatibility with routers, proxies,

firewalls, etc. if your packets are just another TLS stream at the network level. Once you have it, HTTP2 can help you unbundle your assets, so they can be downloaded in parallel. Better yet, you can use its Server Push feature to proactively push down assets before the browser even ask for them. Connections live longer. Headers are compressed, but that doesn't help this situation much. Anyways, suits love HTTP2 because why not, it's 2 right, so you may just sell them HTTP2 and happen to get TLS along the way.

Protocol Relative

If going full https all the time absolutely isn't an option in your organization, you can at least have protocol-relative urls like //feature.js instead of http://cdn.startup.com/feature.js so that when the site's loaded over HTTPS, your js will be to, and likewise for plaintext HTTP.

Certificate Transparency

This is a big deal- if you have ever looked in your OS or browser's trusted store and seen how many CAs are trusted on your machine, it's really really creepy. What happens if a new certificate is minted for your site. Some browsers log new certificates they see, and you can subscribe to certificate transparency logs for your domain. We hooked up a slack notification to Facebook's Certificate Transparency Monitoring Tool so that we get a chat for any new certs.

Certificate Pinning

This one's really great from a security perspective but it's really scary from availability perspective. What if you make mistake operationally in your certificate management and now your site won't load? The key to get this right is to roll it out a little bit at a time. Like five minutes policy expiration, then 30 minutes, then an hour and then 24 hours and then three days a week. Also consider just your employees at first to dogfood it for a bit. Once pinning is baked into production for a while, take that step one step further and get on the pinning preload list in Chrome.

Browser

Extra care needs to be taken in the browser, as you must assume their scrutinized and possibly compromised.

Cookies

Be very sensitive about what you put in cookies. Your customers will scrutinize every single cookie dropped on their machines and so will those customers' customers. So will privacy advocates and privacy regulators. Be very careful about what you put on the cookies and then of course all the normal cookie at hygiene practices- don't store sensitive information, don't trust information them without integrity checks, don't make trust decisions based on it can't, assume they will be tampered with.

HTML5 local storage

Same deal like keep it locked down keep it clean don't trust it. It's like a cookie baking store larger amounts of data.

Content Security Policy

Great idea. Expensive to implement because it break things. Set up a log server, analytics, fix, iterate, spread knowledge.
Few customers use it. Sad!
CSPv3 simplifies implementation so far has limited adoption in browsers.

Subresource Integrity

This is an integrity check for your third party javascript into a customer's page that is loading it. Sounds great, but it ties your hands. The hash changes every time you ship new features! Somewhat incompatible with modern software engineering organizations' continuous delivery.

Self-Hosted Assets

Some customers may care to take their resource integrity one step further and host the assets themselves. Like subresource integrity, the static nature of this approach ties your hands and is incompatible with continuous delivery. This is worse because the customers requiring this, namely FinServ, have super deep pockets and will pay extra for this. It's dirty revenue. You may increase your bottom line, but at a huge opportunity cost when three years from now, they'll only renew if you agree to continue maintaining the specific version of your old code they're still self hosting. Avoid if possible!

Bug Bounty Program

We've tried bug bounty ceilings of 100/500/1000 and 50/500/5000. We call out our javascript client as an area of interest. Not much engagement from independent researchers so far. Soon we'll up our bounty again. We're also reconsidering the bug bounty program going forward, is it worth all of the noise? The traditional consultants and application security assessment have been much more cost effective.

Availability

The internet can fail you, such as when Dynect got DDoSed in October 2016. Have backup plans end to end:

- DNS
- CDN

- Origin
- Build
- Configuration
- Status Page

Looking Ahead

There are still open challenges and areas for improvement, here's a few:

- Anomaly detection - can you detect when your hosted javascript becomes malicious?
- Privacy - you learn a lot being loaded on many top sites, how can you collect data without getting creepy?
- Insurance - how do you underwrite 'pwn once, pwn everywhere' risks?

In Summary

- TLS is essential
- Javascript integrity must be ensured end-to-end
- Avoid giving your customers the opportunity to make insecure choices

Contact

For more information, feel free to ping me or check out my links!

Twitter: @kylerandolph
about.me/kylerandolph

Like this? Optimizely is Growing its Security Team!
optimizely.com/careers