Testimony for The ONC's API Task Force Hearing
January 26, 2015
offered by: David Berlind, editor in chief, ProgrammableWeb

Prelude:

First, I would like to thank the Task Force and the ONC for the honor and privilege of being asked to testify on a matter that is not only of critical importance to the healthcare industry and America's citizenry, but one that is of critical importance to the Internet and all of the people that depend on it.

My answers to the list of questions come from the point of view of an independent observer of the API industry. They do not reflect the interests of any particular party and as you will see, are almost entirely informed by my two-years of research and reporting on real world API security exploits. I believe that this is an important, if not the most important point of view as it bypasses the theoretical and speaks directly to the state of the ever-evolving state of API security and the very real challenges that every organization -- in and outside of the healthcare industry -- will face in the coming years.

APIs are rapidly becoming one of the most important infrastructural layers of the Internet while at the same time becoming a critical component of modern day attacks. They are difficult to secure and determined hackers are extremely tenacious in finding ways to exploit them. Despite what some people --- even experts --- would lead you to believe, there are no silver bullets. That said, when proactively managed and secured, the efficacy of APIs greatly outweighs the risks associated with deploying them.

What follows is my written testimony. It expands, as best I could given the compressed time I had to develop it, on the five minutes of verbal testimony that I will give on January 26th. It is possible that I will make light copy edits to this document between now and the hearing.

Finally, I invite further inquiry by the ONC and the task force. This document barely scratches the surface of a topic that's of grave importance and what I've learned through my research.

1. Does your organization use APIs for apps which are available internally or to third parties?

ProgrammableWeb does not currently offer an internal or external API (though we have an external one planned). However, in the context of this hearing, ProgrammableWeb has much to offer as it has a great deal of experience when it comes to its 10+ years of observing both internal and external APIs.

ProgrammableWeb.com is largely regarded as "The Journal of the API economy" by two primary constituencies; (1) application developers who rely on APIs in the course of programming their applications and (2) API providers -- that is, organizations and individuals that publish APIs (in the case of the former, both internally and externally) for usage by developers.

These two constituencies come to ProgrammableWeb.com for two reasons. The first of these reasons is for our directory of external APIs. ProgrammableWeb maintains the largest independently-run directory of such APIs. APIs are classified according to a variety of criteria

including their category (ie: Travel, Healthcare, Finance, etc.) and for each API we have listed in our directory, ProgrammableWeb maintains a variety of metadata that developers use to compare one API to the next. This metadata includes information about everything from supported protocols and API architectural styles to supported forms of authentication and links to documentation. Over 60% of ProgrammableWeb's traffic comes from Internet users --- mainly developers --- who are researching which APIs to use in their next application. Connected to this directory, ProgrammableWeb offers an alert service. Users can elect to be alerted whenever any of this metadata for their favorite APIs changes.

It is important to note that when we make a record of an API in our directory, we do not offer anything beyond the metadata about that API. In other words, that API cannot be "called" by a developer through ProgrammableWeb, nor does ProgrammableWeb host any of an API's associated assets (ie: documentation). ProgrammableWeb does however provide links that users can click in order to gain direct access to those assets.

In addition to reading our news stories, published daily, about the ongoings of the API economy, the second reason these two constituencies come to ProgrammableWeb is for its prescriptive content. ProgrammableWeb regularly publishes everything from expert commentary to step-by-step tutorials on working with APIs; articles that span the gamut from primers on how to consume popular APIs from companies like Twitter and Apple to how to manage and market APIs if you're someone who has started your journey as an API provider, or is contemplating doing so.

The majority of our prescriptive content is derived from real-world API implementations and is often written by real-world practitioners. ProgrammableWeb's constant contact with the communities it serves and the real-world articles it publishes puts its staff in a unique position to comment on matters of this hearing's nature from a very pragmatic point of view, one that is informed by the broader industry's actual successes and failures.

a. Do you publish your documentation online or make it available to third party developers?

While ProgrammableWeb does not have an API to document, we have catalogued over 14,000 APIs and we are therefore in a position to comment on the generally accepted practices of the industry when it comes to API documentation (both official, and unofficial).

Before continuing, it is worth noting that there is a fair amount of confusion regarding API classifications such as "public vs. private" and "internal vs. external."

In classifying an API's accessibility, ProgrammableWeb currently favors the terminology proposed by Netflix director of edge engineering Daniel Jacobson. Part-way through Netflix's API journey, the company changed gears and, instead of making its API generally available to all developers, it limited API access to selected partners. Other organizations such as Walgreens and ESPN follow a similar model. Jacobson takes a novel approach to describing API availability. Instead of using "internal vs. external" or "public vs. private" to describe API availability, Jacobson uses LSUDs vs. SSKDs.

LSUDs, otherwise known as Large Sets of (mostly) Unknown Developers, describes the sort of API access whereby the API is both external and public. For the most part, this means, in its most broadly accessible offering, there is no barrier, other than the API provider's Terms of

Service (which may require registration and/or payment), to using an API. Generally speaking, the API provider of such an API offers what's known as an API portal through which those interested in the API can, in a sort try-before-you-buy mode, explore the API to better understand the degree to which it may satisfy the developer's needs. This includes the ability to not only view the API's documentation, it also includes sample code and, in a growing number of cases, an opportunity to sample a mock version of the API through an interactive console.

Generally speaking, most API providers who offer this sort of try-before-you-buy experience are targeting an LSUD. The expectation is that many developers will come and self-service themselves through the API portal. From the API provider's point of view, it is most often a low-touch environment with almost no barrier to entry.

In contrast, an SSKD is a Small Set of Known Developers. The API itself may or may not be available on the public Internet (aka: "external"), but it may not be generally available to all developers. Instead, it is only available to selected partners or organizations/developers that the API provider approves of. Whereas Netflix originally took an LSUD approach, it pivoted to an SSKD approach. Today, Netflix's APIs are exclusively available to partners. These are generally well-resourced companies whose offerings are well positioned to expand Netflix's global footprint without compromise to to the end user experience. Sony, which makes Netflix compatible devices like the Sony Playstation, is once partner. **When working with fewer partners, as Netflix does, it can also concentrate its resources on ensuring the best API security practices of those partners**

All this said, here are a couple of important footnotes:

API providers can simultaneously take both an LSUD and SSKD approach. In this environment, developers could very easily be coming to the public portal that has been especially provisioned for them. Meanwhile, the API provider can have a separate program of SSKDs -- known developers who get special privileges or that the API provider works much more closely with.

It should be noted that the majority of API providers that ProgrammableWeb is currently adding to its directory also offer publicly viewable documentation, even to developers who have not yet registered to use the API (if such registration is required). Within the API economy, for APIs that are going to be offered for consumption on either an LSUD or SSKD basis, it is regarded as a best practice to offer such documentation as a part of the marketing effort for an API. In an LSUD context, where an API provider is hoping that tens, hundreds, or thousands of anonymous developers will unlock some new innovation using the provider's API(s), the documentation and associated try-before-you-buy environment plays a very important role in attracting developers.

Likewise, API providers who take an SSKD approach to the market are often looking for qualified partners to help take their business to the next level. Given how any startup could potentially be that next big partner, public documentation plays an important role in business development and partner marketing.

It should also be noted, especially given the subject matter of this hearing, that even when an API provider doesn't offer official documentation for its API(s) (which can sometimes be the case for a variety of reasons including "security by obscurity"), a third party might offer an unofficial form of the documentation. **Unofficial API documentation is sometimes an outcome after a third party has reverse-engineered an API without permission from the API provider.** Third parties are known to publish such unofficial documentation for general

availability on the Internet. A recent example of this involved the APIs for remotely accessing a Tesla automobile.

Furthermore, though not a form of directly documenting APIs, **third parties -- open source developers in particular -- are known to independently publicly publish software development kits (SDKs) for accessing APIs.** SDKs are language/platform-specific (Apple iOS, Java for Android, client-side Javascript, Ruby, Python, PHP, etc.) API abstractions that take some of the pain out of getting an application to work with an API. To the extent that these SDKs can serve as a proxy to the underlying API(s), the documentation of an SDK is essentially a form of documentation of the underlying API(s).

(i) How do you determine who can get access to your API?

Hopefully, the previous explanation of LSUD vs. SSKD sets the stage for answering this question. Under the LSUD approach, an API provider hardly discriminates against who can get access to the API. API access is largely a self-service process and the choice to take an LSUD approach is very much governed by the API provider's business objectives. These objectives often range in purpose; everything from driving additional revenue (when there's a hard dollar cost for using the API) to driving innovation. Generally speaking, (not always), an API provider taking the LSUD approach does not concern themselves who can and who can't get access to the API. There are caveats to this "rule" however. Salesforce.com is an API provider that largely takes the LSUD approach. The company has reported that more than 60% of the transactions with its application infrastructure are API-based. However, before a developer can interact with Salesforce.com's APIs in a production environment, that developer must become a customer of Salesforce.com.

One important footnote to the LSUD approach is that there may be different tiers of developers. For example, there may be a free tier that provides developers with limited access to an API. And then there may be a variety of other tiers, each of which further diminishes the limitations to API access. The important thing to note here is that it's not just about who has access to an API, it is also about the degree to which they have access to the API. This degree of access can impact everything the number of allowable API executions over a given period of time to the scope of access to the underlying resources. For people who are familiar with typical IT infrastructure, this is often described in terms of who can authenticate, and access control (what resources they have access to, based on who they authenticated as).

Like the LSUD approach, the choice to take the SSKD approach is largely one that's driven by the API provider's business objectives. In an SSKD approach, the small set of developers can include external developers, internal developers, or both. But in all cases, the API provider usually knows exactly who has access to its APIs and for what reasons. If it sounds like a far more intimate relationship, it is. For example, Walgreens sees APIs as a way for the company to advance the objectives of its loyalty points program. Through the API, a Walgreens customer's loyalty card can be updated by certain activities; everything from making a Walgreens purchase to a blood pressure check. But the program is exclusive to Walgreens internal developers and a set of partners that Walgreens has chosen to help it log the behavior (e.g.: exercise) of its customers. This approach is very decidedly an SSKD approach whereby all the developers are known and the company's relationship with each developer (internal or external) is very intimate.

Worth noting: in an SSKD approach, it is far easier for the API provider to set and enforce certain security standards to which all developers (internal, partners, etc.) will be held.

(ii) Do they need to be "certified" for privacy or security standards by your organization to use?

This is a great question because as I have become intimately familiar with the ins and outs and intricacies and nuances of API security over the last two years, **I have reached the conclusion that the industry could benefit from the existence of a "Good Housekeeping Seal of Approval" for API security along with a widely recognized and accepted regime to back it up.** I've also given a lot of thought to how such a program would be structured and would operate. Such a certification could apply to both developers and API providers, but currently does not exist in the API economy (more on this later).

Back to the question at hand, I have not heard of an instance where an API provider required a developer to bear the imprimatur of some sort of publicly recognized privacy and/or security certification. That does not mean that such requirements do not exist for certain API providers. They just haven't come to my attention. For example, it's possible that when PHI is involved, certain EHR/EMR providers require developers to satisfy certain HIPAA requirements (or certifiably demonstrate an understanding of those requirements) before they will be allowed access to an API. Such requirements would not easily scale in an LSUD environment. However, it isn't hard to imagine a set of certifiable requirements being applied in an SSKD structured API program where such a program might be more easily managed.

Short of a standard certification program that SSKD-minded API providers (and developers) could turn to for such a requirement, there are no doubt certain API providers who have established certain security and privacy requirements that they themselves check/audit in order for developers to maintain their API privileges. It should be noted that this "need" cuts both ways. Developers who are mindful of their security and privacy of the users of their applications would also benefit from the ability to choose API providers who have certifiably taken measures to protect users whose Personally Identifiable Information (PII) is passing through their API(s).

(iii) Are there terms of use that include specific language for privacy and security?

Such terms of use are very common and are written to address specific patterns as well as non-specific patterns. For example, on the specific-pattern front, Twitter's Developer Terms of Use say the following:

> *"You will not attempt to exceed or circumvent limitations on access, calls and use of the Twitter API ("Rate Limits"), or otherwise use the Twitter API in a manner that exceeds reasonable request volume, constitutes excessive or abusive usage, or otherwise fails to comply or is inconsistent with any part of this Agreement."*

Terms such as these that address rate limiting are not uncommon.

You may ask "What does rate limiting have to do with security and privacy?" First of all, a well-managed API should automatically refuse service once the rate limit for the developer's tier of service has been met. An increasingly less restrictive rate limit is one of the privileges that changes (for the better) as the developer's contracted tier of service improves. But on the security and privacy front, rate limiting (something many consumers encounter when they they get locked out of a service after trying the wrong password too many times) is a common defense against brute force attacks. An example of a brute force attack is when a hacker tries as many user ID/password combinations as it takes to break into a digital account. A good rate limiting policy will

limit the allowed number of failed attempts such that the hacker cannot cycle through enough guesses to compromise an account.

In 2015, compromising, thought-to-be-private, photos of several celebrities including Jennifer Lawrence were shared on the Internet after hackers allegedly penetrated the API for Apple's Find My iPhone service; an API that allegedly had no rate limiting applied to it. Apple never came forward with the low-level details of the incident. But the hackers published the source code -- dubbed iBrute -- that they used to conduct their brute force attack on the API.

Such provisos to protect the API do not just appear in the API's terms of service. Sometimes, they appear in the consuming application's Terms of Service as well. For example, the National Hockey League's branded mobile applications rely on APIs that the NHL does not make available on an LSUD basis. When a user launches the NHL mobile application, the user is immediately confronted with the following splash screens (Terms of Service):

As can be seen from these terms, in an effort to protect its API and ultimately the security and privacy of its users, the NHL is confronting the user with terms that address some very specific scenarios and behaviors as well as some non-specific ones.

It should be noted that an API's Terms of Service do not present a technical barrier to the compromise of the API. Hackers routinely ignore such terms. But they do set the stage for remedy if not legal recourse. For example, one of Twitter's terms says the following:

"Take all reasonable efforts to do the following, provided that when requested by Twitter, you must promptly take such actions...Delete Content that Twitter reports as deleted or expired;"

In 2015, in a highly publicized API economy "incident," Twitter sanctioned the Politwoops service for failing to remove its record of politicians' tweets after those tweets had been deleted. Twitter deemed Politwoops in violation of its developer policy and, as a remedy, discontinued the service's access to the Twitter API (a decision that Twitter has since reversed). While I am not a lawyer, it is clear that terms of services for APIs are not only clearly articulated to support such remedies and legal recourse, but are also evolving to take into account new real-world behaviors that the authors of such terms of services did not originally anticipate.

2. Are there production deployments of these APIs/third party applications using APIs?
The short answer, regarding the more than 14,000 APIs that ProgrammableWeb tracks in its directory, is yes, of course. There are thousands if not millions of deployments of such APIs and depending on the API, everywhere from one to thousands of third party applications using them. There are other API providers such as Google who are testifying today that will no doubt testify to this effect.

For example, one of the most important architectural benefits of an API is the way in which the application or process that consumes an API is "loosely-coupled" from the systems and processes that provide the API. One key benefit of such loose coupling is that, so long as the parameters that an application uses to interact with an API remain unchanged, the API provider is free to make changes to the underlying systems and processes that provide the API. For

example, much the same way a power utility can substitute nuclear-generated power for coal-generated power without interrupting the functionality of common household appliances, an API provider can change the operating system that provides their APIs from Linux to Windows without disrupting the applications that consume that API.

This architecture has proven to be a major boon to the development of mobile applications, the majority of which rely on the loose-coupling principles of APIs to interact with their servers or, as some would say "to phone home." When you consider the sharp proliferation of mobile applications (including the aforementioned NHL application) since the iPhone was first introduced in 2007, the grand majority of them consume APIs, thereby giving you some idea of the degree to which APIs have been put into production and the number of third party applications that rely on them.

In fact, when ProgrammableWeb was initially founded in 2005 and for many years after, it kept a record of all the applications -- then known as "mashups" -- that consumed Web APIs. But with the advent of app stores and mobile applications, it became quite evident that there was no way that ProgrammableWeb could scale its directory to match the growth of the various app store inventories. The "mashup directory" is therefore a part of the ProgrammableWeb directory structure that we no longer actively maintain because it's not nearly representative of the true number of "mashups" that are currently available through app stores.

One additional point worth raising here is that the so-called "Internet of Things" will take this proliferation of APIs and the third-party applications that rely on them to an entirely new level. There is really no such "thing" (including a great many medical and fitness devices) as a "thing" on the "Internet of Things" that doesn't have an API so that applications and other processes like microservices can interact with it. When you start to consider how every appliance, light bulb, car, and even the buttons on your shirts will be connected to the Internet, it isn't difficult to imagine the orders of magnitude to which APIs and the third-party applications that work with them will be put into production.

3. What are the perceived and actual privacy and security concerns or barriers to the adoption of APIs?

The first thing to consider when answering this question is how **APIs are, in some ways, gaining a reputation for becoming a wonder drug of the type that the Web was in the mid-90's.**

Back in the mid-90s, as the World Wide Web caught-on --- thanks in a large part to the tech and mainstream media --- as means for businesses to very cost-efficiently communicate with their customers, it wasn't long before every business was racing to attach one or more Web sites to the Web. Soon, it was inconceivable that a business could exist without also hanging its shingle on the Web. Today, APIs are the new Web. **And thanks to the way the tech and mainstream media are painting a picture of APIs as the new must-have in order to not only conduct, but to reinvent business for participation in the new economy, businesses are once again racing to put API technology into production (and rightfully so in a great many cases).** As the virtues of APIs are extolled in business media such as Forbes and Fortune, CEOs are scrambling their IT teams to beat the competition to the API-punch.

However, while there is no shortage of information about this API gold rush that's not to be missed, there is a distinct lack of information about the security and privacy risks that

go with APIs and the degree to which a so-called "rush-job" could endanger the same digital assets that the API was meant to leverage.

I want to be clear here; every digital technology that the human race has found to be both promising and useful has also come with its risks. No such technology has proven to be infallible. But we, as people, very often come to the conclusion (one that I agree with), that the efficacy of the technology greatly outweighs the risks associated with using it. I believe this to be the case with APIs, or I wouldn't be here at this hearing, or in this job as the editor in chief of ProgrammableWeb.

That being said, as thousands of companies rush to put their APIs on the Web, very few if any fully appreciate the difficulty in securing them. In other words, **the widely-perceived state of API security reflects a belief that if you rely on well-known Internet, Web, and API security standards to provision your API, then your API will be secure. However, in practice, this dangerous assumption has not borne out to be true** and here are some reasons and real-world examples why:

>> **Since 2014, many of the biggest Internet companies on the planet -- the ones with nearly unlimited financial resources to devote to API security (in other words, they can employ the very best experts) -- have either fallen prey to, or discovered a major API vulnerability in their API(s).** This includes companies like Google, Apple, Facebook, Pinterest, and Snapchat. In the last month alone, both Trend Micro (a company devoted to security) and Verizon were the latest companies to have experienced highly publicized API security challenges. In my opinion, security oversights and/or an API design flaws were major contributors to all of the vulnerabilities. In other words, "human error." This begs a very important question: **If these companies, with the deepest pockets to employ the best experts, are experiencing challenges in securing their APIs, how can less resourced organizations be expected to do the same?** This question matters mostly when API security is a forethought, long before anyone sits down to design an API. In a great many cases, API security is an afterthought.

>> **When mobile applications are in use -- which involves a great many API cases -- the majority of the API secrets that are (a) shared between the mobile application and the API and (2) presumed necessary to secure the API and any communications with it, are easily discoverable even when standard security technologies like HTTPS are thought to have secured those communications.** Not only are there are a variety of methods to expose these secrets, there are a variety freely downloadable tools -- even ones that we can download to our smartphones -- that require almost no expertise to operate. The only topic I currently speak about publicly (at conferences, etc.) is API security. During these talks, I show examples of how easily this is done.

>> One of the most promising aspects of APIs is how they scale. APIs make it possible for one application to repeat its instructions to another application with extraordinary speed. As such, APIs are ready made for hackers looking to do a lot of damage in a short period of time; which is invariably their M.O. **The pupils in hackers' eyes swell at the sight of potentially exploitable APIs.**

>> **Depending on the importance of the objective, most successful real-world API exploits involved a very multi-dimensional attack that is very difficult to defend against.** For example, in October 2014, when hackers stole the security tokens (known as Oauth tokens) that it made it possible for them to impersonate thousands of Twitter and Facebook users, the

attack involved phishing attacks on two separate companies, the unauthorized access of a Github source code repository, the penetration and subsequent “screen-scraping” of a customer support application, and then finally thousands of unauthorized Twitter and Facebook posts (whose links very likely infected users who clicked them with malware). The damage was done before any of the involved parties knew what hit them. In other words, the juicier the carrot, the more sophisticated the series of orchestrated exploits to get to it. Hackers will literally stop at nothing and are usually doing this while multiple vulnerabilities are left unguarded.

>> **API security, like many forms of security, is a cat and mouse game. Just when you think you’ve got a security scheme that will keep the bad guys out, they find another way in (which invariably sends everyone back to the drawing board to close that loophole).** Case in point? In the aforementioned October 2014 attack, the hackers gained unauthorized access to Oauth tokens that in turn allowed them to impersonate the end users that those tokens represented. In recent months, the Internet Engineering Task Force (IETF) has been moving quickly to ratify the necessary security standards that will require a program to prove that it has the right to use an Oauth token on behalf of some user. This is called “proof of possession” or “POP.” Had this standard been baked into commonly used API software configurations prior to October 2014, it might have prevented the aforementioned attack. **This issue also speaks to the lag time between the ratification of new security standards and the time it takes for those standards to take root in the solutions that API providers use to manage and secure their APIs.**

a. How can these risks be mitigated/how are you addressing this? In my humble opinion, the risks can be mitigated in a variety of ways. First, it is important to recognize that API providers generally take one of two approaches when it comes to providing their APIs; they either build the majority of their bespoke API infrastructure from scratch or they rely on canned API management and security solutions that pack many of the better known API security approaches behind the click of a mouse. However, as said earlier, the API attacks that we have observed in the real world are sophisticated, multi-dimensional attacks that involve more than the API infrastructure itself and, as with most sectors of tech, even the most advanced solutions are sometimes out of step with the most freshly baked security standards.

As such, the proliferation of a constantly evolving set of best practices --- an API security checklist or cookbook if you will --- for not just securing APIs, but their adjacencies as well, can inform the key stakeholders on how to maintain the best possible API security. For example, in the aforementioned incident where the unauthorized penetration of a Github source code repository was one link in the chain that led to the ultimate attack, requiring two-factor authentication to access that source code repository would have stopped the hackers dead in their tracks.

When I say constantly evolving, an example of this would be how this month’s white-hat discovery of an API-related exploit of the user ID/password management service LastPass would immediately inform the checklist.

If such a cookbook were based on both theoretical and real-world incidents (in real time), the architects of both bespoke and canned API management solutions would have a comprehensive resource to turn to in order for their implementations to cover a majority of the beachfront.

Additionally, the same checklist could be used as the basis of some sort "Good Housekeeping Seal of Approval" whereby the checklist would serve as the basis for independently conducted audits in order to earn the seal.

In researching the majority of the real-world API attacks that have taken place over the last two years, I have begun to formulate such a checklist while contemplating a seal of approval and what role ProgrammableWeb could play in maintaining such a program. But so far, it's nothing more than an idea.

Another way to protect APIs is to make sure a majority of their interactions are taking place behind a firewall (as in making them server to server interactions vs. client to server). But this is an architectural approach that doesn't easily apply to a great many situations.

Today, the most favored approach to closing the exposure of API secrets when a mobile application is involved is known as "certificate pinning." It is a technique that not only breaks the tenants of the DNS (which is partially responsible for making the Internet operate the way it does), it significantly erodes the aforementioned API benefit of loose coupling.

Finally, and this is true for every industry, (1) something must be done to ensure that white-hat activity does not end in criminal prosecution and (2) it is beholden upon the various industries and the companies in them to establish bug bounty and disclosure programs that encourage the type of white-hat research that ultimately results in a much better-protected infrastructure.

4. How to improve consumer experience with the third party apps using the APIs?
To me this question is, how do you instill confidence in consumers that their applications are safe to use. It is this very question that I asked myself and that provoked me to consider the idea of a Good Housekeeping seal of approval and all the elements that would make such a program successful. They are too long and detailed to enumerate here but I would be happy to go into more depth at the future request of the task force.

5. Are there third party certifying authorities in non-healthcare industry that we can leverage? I don't think there are third party certifying authorities that can be leveraged. But there are examples to learn from and I think TrustE and NIST's Green Button are two of those. In fact, ProgrammableWeb recently endured a TrustE audit and, as a result, we had to make several adjustments to the way our Web site's user experience works.