WHITE PAPER | SEPTEMBER 2014

Protecting Your APIs Against Attack and Hijack

Secure your enterprise applications for mobile, the cloud and open Web.

ca technologies

Table of Contents

Executive Summary

Challenge

The application programming interface (API) is an emerging technology for integrating applications using Web technology. This approach is exploding in popularity because it builds on well-understood techniques and leverages some existing infrastructure. But it is a mistake to think you can secure APIs using the same methods and technology with which we secured the browser-centric Web. APIs are fundamentally different from websites and have an entirely unique risk profile that must be addressed.

Opportunity

The best practice for API security architecture is to separate out API implementation and API security into distinct tiers. Under this model, an API developer can focus completely on the application domain, ensuring that each API is well designed and promotes integration between different apps. CA API Gateway provides the API security administrator with complete control over access control, threat detection, confidentiality, integrity and audit across every API the organization publishes.

Benefits

CA API Gateway (formerly CA Layer 7 API Gateway) features a policy-based security model that is easily tailored to accommodate different security requirements for each API. It offers core policies that can be easily shared across sets of APIs to create a consistent basic security stance, as well as the ability to specialize on an API-by-API basis to meet the specific needs of a particular application. It also integrates easily with existing identity systems (LDAP, IAM etc.) and operational monitoring systems.

Section 1

New Technology, Old Threats

APIs may represent a relatively new technology, but they are susceptible to many of the same risks that have plagued computing since the early days of the Internet. SQL injection, for example, has its roots in the beginning of client/server programming. You would expect that, by now, developers would be well versed in countermeasures for coping with such a mature attack. But SQL injection migrated easily into Web apps and it is now seeing a significant resurgence as more organizations publish Internet-facing APIs linked directly into their application infrastructures.

APIs are windows into applications and—as with any window—an API can easily be misused. Well-designed APIs have the quality of self-description; a good developer should be able to intuit how to use an API simply by inspecting its URL, the input parameters and any returned content. However, this same clarity often exposes the underlying implementation of an application—details that would otherwise be obfuscated by delos of Web app functionality. The transparency an API provides into the internal structure of an application often gives hackers dangerous clues that may lead to attack vectors they might otherwise have overlooked.

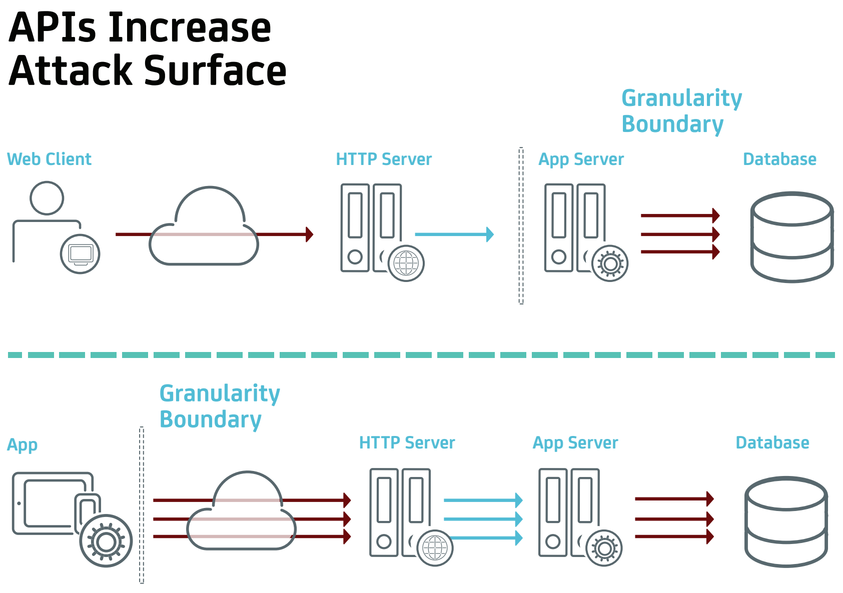
Not only do APIs put applications under the hacker microscope, they also offer greater potential for nefarious control, by increasing the attack surface on client applications. APIs give client-side developers—both legitimate developers and potential system crackers—much more finely-grained access into an application than a typical Web app. This is because the granularity boundary for calls to backend tiers moves from relatively secure internal tiers (those that reside safely in a DMZ) all the way out to the client application residing on the Internet. This is shown in Figure A.

A typical Web app consolidates finely-grained operations—such as calls to a data tier—behind simple Web pages. On the traditional, HTML-oriented Web, interactions were limited to basic form interactions. This reduced the range of potential interactions and—in doing so—reduced overall exposure, particularly if the application properly sanitized its inputs.

APIs, in contrast, move this granularity boundary all the way out to the client application itself. This offers a hacker considerably more avenues to exploit. Rather than having one simple form acting as a proxy for many internal calls to backend resources, an API-driven application may individually make all these calls on its own and any of them may expose vulnerabilities. This growth in attack surface increases the risk an organization must manage as it publishes APIs to the greater Internet.

Figure A:

APIs move the function granularity boundary out to the client, giving a hacker much greater attack surface to work with.



Three broad risk categories

It is easy to become dismayed by the range of potential attacks against APIs. Since every API is unique, each instance carries unique risk based on its underlying implementation. This would seem to make API security impossible. Fortunately though, most individual attacks against APIs fall into one of three broad categories:

1. Parameter attacks exploit the data sent into an API, including URL, query parameters, HTTP headers and/or post content.
2. Identity attacks exploit flaws in authentication, authorization and session tracking. In particular, many of these result in bad practices from the Web migrating into API development.
3. Man-in-the-middle attacks intercept legitimate transactions and exploit unsigned and/or unencrypted data. They can reveal confidential information (such as personal data), alter a transaction in flight or even replay legitimate transactions.

By understanding these broad categories, we can begin to design an effective mitigation strategy against API attacks. An effective API security strategy should even guard against unforeseen future attack vectors. Each of the three attack categories is described in detail below.

Parameter risks

The classic parameter attack, as already observed, is SQL injection. Parameters designed to load legitimate input into a database query can often be manipulated to radically change the intent of an underlying SQL template . Script insertion exploits systems that ultimately interpret submitted content as a script. The classic example is a snippet of JavaScript submitted into a posting on a Web forum. If this is not intercepted on the way in (and fortunately, virtually all forums now do detect such attacks), any subsequent user reading the post will have the script execute in their browser, potentially hijacking session tokens or other important information. But script injection can also take place server-side, where poorly-designed applications evaluate user-submitted data and respond to encapsulated script.

Bounds or buffer overflow attacks are also parameter risks. These attacks attempt to exploit a system by providing it data outside of the expected range or type, which can lead to system crashes, or offer access to memory space. This is the classic attack against C programs that was so popular in the 1990s but still exists today.

Parameter attacks all result from developers not carefully restricting inputs to a narrow range of anticipated types. APIs are particularly susceptible to these attacks because their self-documenting nature often provides hackers with insights into how an underlying system uses a parameter. Unlike Web apps, where the ultimate use of a parameter is often obfuscated or even completely masked behind an HTML veneer, APIs may clearly identify a parameter's underlying meaning. This is particularly common if the API is automatically generated from underlying application code, as meaningful internal names are typically directly mapped to external API parameters.

Identity and session risks

Hackers have long used forged or stolen credentials to exploit applications on the Internet. In this respect, APIs simply provide another avenue to apply the same attacks. But APIs also open unique attack vectors leveraging identity. A number of these attacks exploit common bad practices originating in the Web app development community. As Web developers move into API development, they often bring bad habits from the conventional Web. Other attacks result from widespread confusion about how APIs differ from traditional Web app development.

Many applications publishing APIs require clients to use an API key to access to their functionality. The name API key, however, is a bit of a misnomer, as it implies that the key is unique to one particular API. In fact, an API key is associated with a client-side application and may be applied across the entire range of different API calls an app might perform. Typically, the API key is an opaque identifier issued to a developer and embedded within their application. Thus, it identifies a particular application but not a particular instance of an application, as this key is common to every copy of the app.

It sounds straightforward but—in practice—API keys are the source of great confusion. And this confusion is the source of a new API attack vector. An API key does not identify a user; nor does an API key identify a unique instance of an application. Moreover, API keys are simply not authoritative, as the key itself is typically obfuscated in compiled code and so subject to extraction by any skilled developer. ("Reuse" of existing application API keys is a common developer trick to deal with API managers, such as Twitter, that restrict the distribution of new keys.)

API keys should never substitute for user credentials when authorizing access to APIs. An API key must be treated as a non-authoritative tracking mechanism to provide basic operational metrics (such as relative load coming from different applications). It should not be the basis for auditing that might lead to chargeback, simply because it cannot be trusted to be handled securely on the client, like a password. It is simply too easy to fake an API key, so its existence cannot be relied upon for audits that matter.

Unfortunately, too many apps make cavalier use of API keys, as if they were securely-stored shared secrets. This informality provides hackers with a simple new attack vector to exploit. If the security model of an application assumes an API key is unique to a user, or is a secure and authoritative proof of a particular client app, the server application is vulnerable to misuse.

Session management is another area of considerable confusion in the API community. Developers coming from the Web app world are loath to include formal credentials on every transaction, as this is not something you would ever consider on a conventional Web site. On the regular Web, browsers maintain session using cookies or other opaque session IDs. Unfortunately, APIs rarely take a consistent approach to session management, leaving developers to create their own, sometimes of dubious quality.

Arguably, OAuth fills this gap and it is the preferred approach for API development. But OAuth is hard to apply well and still relies on protection of critical keys such as access tokens and refresh tokens both on the client and in transit. It is a promising technology but the application of OAuth remains very challenging. Incorrect configuration and application are common problems that increase API risk.

Man-in-the-middle risks

APIs are subject to increased risk from man-in-the-middle attacks when the transmission is not encrypted or signed or when there is a problem setting up a secure session. If an API does not properly use SSL/TLS, all requests and responses between a client and the API server can potentially be compromised.

Transmissions may be altered in transit or replayed, and confidential data—such as personally identifiable information, session identifiers etc.—may be captured. Even those APIs that use SSL/TLS are at risk if this is improperly configured on the server side, if the server is subject to downgrade attacks on ciphers or if the client is not properly validating the secure session (such as full certificate validation, following trust chains, trust of compromised or suspect CAs etc.)

Some of this risk is the result of persistent cultural issues. In the Web app world, SSL is usually only engaged on a handful of "important" transactions, such as credit card or password submission. Certainly these data need protection but so do the session keys that accompany other run-of-the-mill transactions following an initial sign-on. Tools such as Firesheep, which easily intercept cookies in unencrypted Web transmissions, dramatically illustrate the risks associated with this approach.

This practice is largely an historical artifact left over from the days when SSL was computationally expensive to perform on servers. But Google has demonstrated persuasively that, on contemporary servers, the additional computational load arising from SSL is largely insignificant.

Despite this, many developers leave APIs open and unprotected simply out of habit. In fact, the original version of OAuth included signatures applied across query parameters as a way of ensuring integrity in transactions. OAuth 1.0a used these signatures to guard against man-in-the-middle attacks that might alter parameters in flight or hijack access tokens for use in other transactions. This was a half step in the right direction; it acknowledged that a risk exists but only mitigated for a subset of the potential API attacks (for example, it ignores confidentiality issues completely).

In practice, developers found it very difficult to apply signatures consistently in OAuth. Version two of the specification made signatures optional in acknowledgement of this complexity, instead mandating use of TLS/SSL to protect bearer tokens. As a side effect of this change, the new solution addressed confidentiality issues. However, this solution still assumes that SSL/TLS be set up correctly—and this is something that is decidedly more complex than simply appending "s" to http in an API URL.

Section 2

Mitigation Guidance

Although APIs are susceptible to a broad range of attacks, application of just five simple mitigation strategies will allow an organization to securely publish APIs.

Validate parameters

The single most effective defense against injection attacks is to validate all incoming data against a whitelist of expected values. The most practical approach to this is to apply schema validation to all incoming data against a highly-restrictive data model. Schemas should be composed to be as tight as possible, using typing, ranges and sets whenever possible. Unfortunately, the schemas produced automatically by many deployment tools often reduce all parameters to simple strings and are ineffective in identifying potential threats. Hand-built schemas are much preferred, as developers can constrain inputs much better, based on their understanding of what data the application expects.

Posted, structured data should be validated against a formal schema language. XML content has the advantage of a richly-expressive XML Schema language that is highly effective in creating restricted content models and structure. However, applications can validate JSON-based content using an emerging JSON schema language. This technology may not be quite as rich as XML schema but—like JSON itself—the schema language is far simpler to compose and understand than XML, giving developers a decided advantage.

Query parameters generally map to simpler types and can therefore be validated using simple textual models. Regular expressions are an excellent approach for building a query parameter schema that is highly constrained but widely understood.

Apply threat detection

Threat detection is generally an exercise in blacklisting risky content, such as SQL statements or SCRIPT tags. The challenge is to apply this effectively. It is easy to scan incoming data for text such as SELECT; however, it is much harder to avoid throwing false positives when this character sequence legitimately occurs. It is important to be able to tune your security scanning appropriately for the API at hand.

Virus detection should also be considered on encoded content. POSTed binary content can potentially mask executable content. APIs involved in file transfer should definitely decode base64 attachments and submit these to server-grade virus scanning. Many commercial anti-virus vendors offer a server-based solution with a simple ICAP-compliant API. An application should submit content to this before it is persisted on any file system, where it could potentially be subject to activation as a virus.

Finally, remember that message size or complexity can itself be an API threat. Very large messages, heavily nested data structures or overly complex data structures can all be effective denial-of-service attacks against an API server. It is risky to simply start processing parameters in an application without first validating that the content is within an acceptable range.

Turn on SSL everywhere

Make SSL/TLS the rule for all APIs. In the 21st century, SSL is not a luxury; it is a basic requirement. Adding SSL/TLS—and applying this correctly—is an effective defense against the risk of man-in-the-middle attacks. SSL/TLS provides confidentially and integrity on all data exchanged between a client and a server, including important access tokens such as those used in OAuth. It optionally provides client-side authentication using certificates, which is important in many high-assurance environments.

However, despite it being much simpler than message-level security methods such as WS-Security, SSL is still subject to misuse. It is easy to misconfigure on the server and research has shown that many developers do not properly validate certificates and trust relationships when applying it . Spend time auditing both server-side and client-side configuration and use.

Separate out identity

User and app identity are separate things. Developers must recognize the limitations of application ID credentials—such as API keys—and understand the right context for using these. Server-side developers should clearly separate out user identities and API identities and potentially even consider authorization based on a broad identity context, which can include the entire set of details such as user, application, device (e.g. a particular mobile phone), location, time etc.

OAuth is quickly becoming the accepted method of authorization for APIs but it is important to realize that OAuth remains a complex and confusing technology. Although OAuth has, in some respects, become simpler as it evolved from version 1.0a to 2.0, its scope has increased dramatically. This has created considerable confusion over how and when to apply it. It has also created significant interoperability problems. Developers should defer to the basic, well-understood use cases and always use existing libraries rather than trying to build their own.

Use proven solutions

A basic rule of security is: Do not invent your own. APIs are no exception to this rule. There is no reason to create your own API security framework. Excellent security solutions already exist for APIs, just as they do for the Web. The challenge is to apply these. Libraries for form-based parameter validation in Web apps are common, so consider these in your API architecture. Similarly, there are many good OAuth libraries available for most common languages.

Another proven approach is to separate out API security from the application that publishes the API. This allows app developers to focus completely on application logic, while a dedicated security expert focuses on applying security policy consistently across all APIs. This approach is described in the next section.

Section 3

Get Complete Control over API Security

Secure API architectures

The best practice for API security architecture is to separate out API implementation and API security into distinct tiers. These separate tiers emphasize that API design and API security are essentially different roles requiring different expertise. This is a very logical *separation of concerns*, one that focuses expertise on the right problem at the right time.

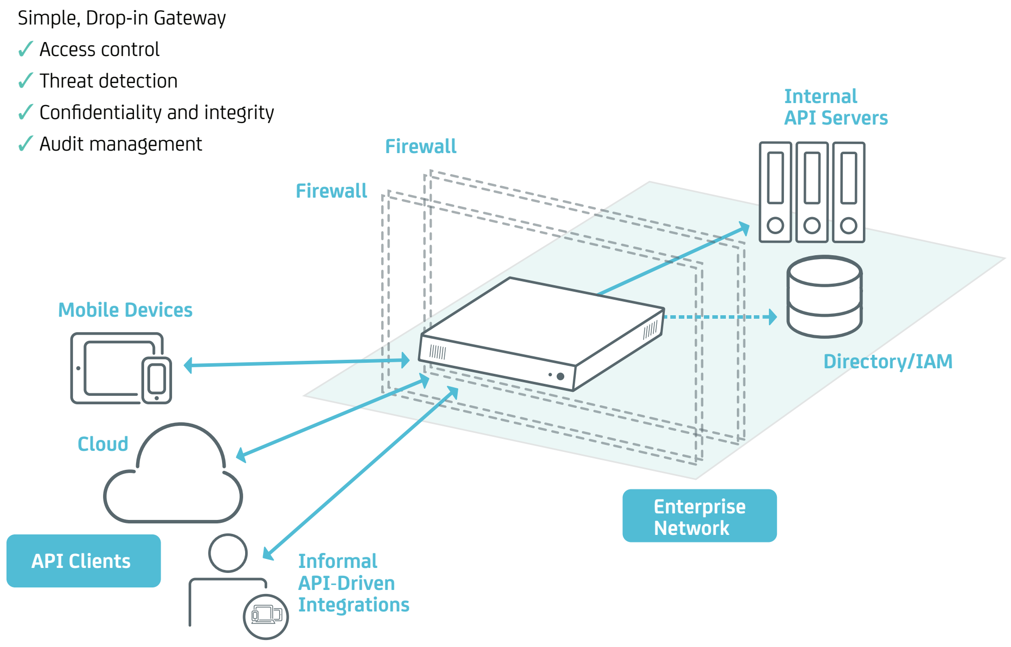
Under this model, an API developer can focus completely on the application domain, ensuring that each API is well designed and promotes integration between different apps. This releases the developer from responsibility for securing the published API, as this will be performed by a dedicated API security professional.

The API security expert is responsible for applying security policy consistently across the organization. Their focus is on identity, threats and data security—in other words, each of the issues and threats highlighted in this paper. It is vitally important to give anyone performing this critical role the right tools for the job, as they will need to separate out security from the actual API implementation.

CA API Gateway is the right tool. This is a hardened appliance, which is provided in either a physical or virtual form factor and is generally deployed in an organization's DMZ, as shown in Figure B below. It is the secure proxy between the internal application and the outside Internet. CA API Gateway provides the API security administrator with complete control over access control, threat detection, confidentiality, integrity and audit across every API the organization publishes.

Figure B:

CA API Gateway creates a separation of concerns between server-side API development and security across the API.



CA API Gateway features a policy-based security model that is easily tailored to accommodate different security requirements for each API. It offers core policies that can be easily shared across sets of APIs to create a consistent basic security stance, as well as the ability to specialize on an API-by-API basis to meet the specific needs of a particular application. It also integrates easily with existing identity systems (LDAP, IAM etc.) and operational monitoring systems.

CA API Gateway provides the security administrator with complete control over all aspects of API security, including:

- Threat detection, including SQL injection, XSS, CSRF, message size, etc.
- Confidentiality and integrity, including limitations on cipher suites, advanced ciphers based on technology such as ECC digital certificates, message-based security etc.
- Message validation, including XML and JSON schema validation
- Authentication support, including basic credentials, API keys, OAuth, SAML, OpenID Connect, Kerberos, x509 digital certificates, etc.
- Integration with most common identity and access management (IAM) systems, including generic LDAP, Microsoft AD, Tivoli Access Manager, Oracle Access Manager, CA/Netegrity, etc.
- Rate limiting and traffic shaping of transactions against any model
- Rich policy-driven audit and eventing, including the ability to trap events based on custom criteria and to integrate with existing security and management infrastructure

Section 4 :

Conclusions

It has been said that those who do not learn from history are doomed to repeat it. API security seems to bear this out. It is important to recognize, however, that APIs add unique new threats that must be addressed. The real insight we should take from history is that application developers will leave security to the end of the project, rushing implementation, if they get to it at all. This behavior is difficult to change. It is time to take a different approach.

Separating out API development and API security implementation is the best practice to follow in order to create a secure API strategy. CA API Gateway gives security administrators the tools they need to secure an organization's APIs.

Connect with CA Technologies at ca.com

1 Best illustrated by XKCD: http://xkcd.com/327/

2 For an excellent study of common problems in client-side SSL/TLS use around APIs, see: http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf