WHITE PAPER

Defend Against Bots from the Beginning

Best practices for developers to create resilient web and mobile apps

You're a developer, responsible for your company's mobile and web apps. It's noon on a Friday and you get a call from Security Operations. There's a bot attack happening against a critical API endpoint, threatening the confidentiality and availability of your system, and SecOps cannot stop it with the tools they have. Is your weekend ruined? Is this even your problem? Unless you anticipated this scenario from the start, the answers to both questions could very well be yes.

Threat actors using automated scripts — bots — to abuse web applications are a big problem, but they can be a much bigger problem if your application isn't developed with them in mind. Failing to account for bots in your design decisions could mean frequent and significant revenue-impacting events, like outages and periods of poor user experience, along with the sleepless nights, missed personal time, and the additional expenses involved in all-hands-on-deck incidents.

There are proactive steps you can take to avoid such instabilities. We recommend these 10 guidelines to help developers create apps that are less likely to be harmed by bots and to create anti-bot security measures that are more likely to work.

TL;DR

1. Cache everything, everywhere you can
2. Document "sensitive operations," like logins, and the workflows that call them
3. Identify and document the clients that interact with your app
4. Make it easy to identify and update your mobile apps
5. Make it easy to identify when a sensitive operation takes place
6. Stay in the same cookie space
7. Eliminate sensitive operations that happen prior to authentication
8. Reduce the number of sensitive operations in general
9. Use normal web browsers, outside your app, for authentication
10. Make it easy to distinguish failed logins from successful logins

The 10 guidelines

1. Improve caching

As part of a comprehensive strategy to mitigate the effects of bots — while also improving the performance and offload of traffic in general — organizations should increase caching wherever possible. Anything that can be cached should be cached, and it should be cached anywhere it can be cached. Making use of the ability of your content delivery network (CDN) to cache as much as possible will eliminate most bot problems that involve distributed denial of service (DDoS), as those cached requests will never hit your app server.

With Akamai, you can go a step further: Static resources like fonts, images, and Cascading Style Sheet (CSS) files can be served from our cloud storage service, Akamai NetStorage. Serving content from NetStorage ensures origin processing and bandwidth offload even when an object isn't cached — and the best part is that you probably have a large allocation of NetStorage space just by being an Akamai customer. This service is especially important for websites for which caching is difficult due to a large number of assets; for example, a digital commerce site with hundreds of thousands of product pages, producing a tremendous amount of "long-tail content" that never gets cached (or, if cached, never gets pulled from the cache). By putting these assets on NetStorage, you effectively introduce another layer of caching and completely offload that content from your origin.

Even dynamic, private content (constantly changing data unique to individual users) can probably be cached for at least a little while in the client, even if it can't be cached for long periods at the CDN layer. That reduces the number of requests that legitimate clients make and helps reveal bots that come in at higher volumes.

2. Identify and document "sensitive operations" that are particularly ripe for automated abuse

Being mindful of what you're designing is critical to ensuring that your organization takes appropriate precautions and hardens defenses during the build phase, so you'll be prepared for any trouble that arises in production. The following common operations on a site are likely to be subject to automated abuse and should be considered **sensitive operations**.

Sensitive operations

- Logins
- Account creation/verification
- Password resets
- Real-time inventory or price lookups
- Store locators
- Search
- Contact forms
- File uploads
- Authorization, validation, or balance check operations for payment cards, financial accounts, or customer loyalty reward programs
- Any other legitimate operation that could cause significant origin load or incur other costs

Every sensitive operation should to be inventoried, and their workflows need to be fully documented, including the method and URL(s) of the page(s) with which the user interacts prior to making the operation, how the operation is invoked (such as through a form submission or AJAX), and how frequently the operation is called during a typical user session.

As workflows become scattered, it becomes increasingly difficult to inventory sensitive operations and to document the workflows that invoke them. For example, imagine a login operation that could potentially be invoked from dozens of different domains to dozens of different URLs. If all logins were always and only ever made against a particular URL, it would be much easier to document this and, later, to target it for protection. IT teams should do that wherever possible. With one caveat: There are instances when it’s a good idea to have different login workflows for different clients, as discussed in guideline 6.

If many different URLs must be used (as in the case of different brands or country TLDs that each have their own domain name but are all using the same backend application or framework), at least be consistent from one domain to the next so that you can say things like “Users will always GET /LoginForm before they POST to /Login” regardless of domain name.

Additionally, it’s also critically important to keep in mind users who come in from “deep links” like search engine results or their own bookmarks when documenting these workflows. Generally speaking, sensitive operations should not be able to be directly invoked by a browser when simply opening a link (see guideline 7 for more details).

3. Identify and document the clients using your app

Web browsers, whether on desktop or mobile, are virtually identical in terms of their ability to interpret HTML, CSS, and JavaScript — as well as in their need to request those resources in each new session. This is not so for other clients, like **native mobile applications** (NMAs) or most automated tools like systems monitors, partners making API calls, or even malicious bots. Because all this is incredibly important to how bot management works, the known legitimate clients of the above-noted sensitive operations need to be documented also, so that SecOps can better identify and mitigate automated abuse.

For each of your sensitive operations, document whether requests for them will come from web browsers, nonbrowser clients, or both. Examples of these nonbrowser clients include NMAs, as well as kiosks, account aggregators (common in financial services), business-to-business communications with partners and resellers, or anything else that is not an actual web browser being used by a human. Also document how any nonbrowser clients are identified, such as a specific User-Agent string or a particular IP range. If IT professionals know that there are going to be legitimate automated or other nonbrowser clients making requests for sensitive operations, but have no ability to reliably identify those clients before they make their requests, that's a red flag. Those professionals need to reconsider how they're approaching the objective of their design, as it may be inherently insecure.

4. Make it easy to identify, validate, and update your clients — especially mobile apps

You should have a consistent way of identifying requests that come from clients that you control — this is particularly important for any NMA that you develop. Every request that a certain client makes should have the same User-Agent string, and that string should contain information about the platform (iOS, Android) and app version (using numbers; e.g., v7.3.2). Further, that User-Agent pattern should be consistent among different platforms and among different versions. Although having a consistent, identifiable User-Agent string isn't itself a guard against bot operators who can easily spoof those strings, having this will help SecOps identify aberrant traffic patterns of clients that claim to be a particular version running on a particular platform.

Examples of good User-Agent strings for an NMA	Examples of bad User-Agent strings for an NMA
MyApp/Android v1.2.3.4	**OkHttp 2.0:** Generic user agent for Android/Java HTTP clients
MyApp/iOS v5.6.7.8	**MyApp/Version.A.B7 Android/iOS:** No way to distinguish platform; version string characters make it more complicated to do programmatic analysis of traffic logs

Additionally, include some way of forcing an update on these clients, whether via a simple request to update or by breaking the app until the update happens. The client should check on startup to see the current available version (or at least the minimum version you want interacting with your APIs) and compare that with its own version to determine how to proceed. This functionality is critical in situations where new app deployments need to happen as a result of evolving security requirements, such as the addition of Akamai Bot Manager.

Finally, you need to be sure you're making use of the tools that Google and Apple have built into their operating systems and app stores that help you validate genuine app installs. This is done via Android's Play Integrity API, which also requires you to identify sensitive operations, and Apple DeviceCheck.

5. Create clear, distinct operation identifiers

For some applications, different operations involve different URLs — for example, an unauthenticated operation to fetch information about the service's status could happen at /UptimeStatus while an authenticated operation to check on a bank account balance could happen at /AccountBalance. For other applications, the same URL can be used for both operations, along with many or all other operations involved in the application, and the origin-side application will distinguish between different operations based on some other identifier in the request, such as a header, query, or body parameter. This is common for APIs used with both in-browser single-page applications (SPAs) as well as NMAs.

However your app identifies the operation, whether by URL or not, use one operation identifier for a sensitive operation being performed by web browsers and a different operation identifier for that same sensitive operation being performed by NMAs and other nonbrowser clients. For example, if the operation identifier is a URL, a browser login could be done at /BrowserLogin, while an NMA login could be done at /MobileLogin.

This doesn't mean making a different URL for every different client, of which there could be hundreds or even thousands. It's generally enough to have just one URL for web browsers and another URL for anything else — but, if you do want to go the extra mile, you can make one identifier for web browsers, one for clients that you control (such as NMAs that you develop), and a third URL for clients that you don't control (such as partner API clients or authorized third-party NMAs). Again, this allows for SecOps to better identify and mitigate automated abuse while reducing the risk of blocking legitimate clients and requests, and really only needs to be done for sensitive operations, not your whole site.

Additionally, organizations should also have different identifiers for authentication using human-known credentials (username and password) versus authentication using other types of credential artifacts (like OAuth's automatic reauthentication operations using refresh tokens).

In any case, the identifier should be as obvious and simple as possible so that sensitive operations can be efficiently targeted for protection. It won't be efficient for SecOps to have to look at two different request parameters, or a parameter that is nested 10 levels deep in a JSON array, to tell if a request is a login or not.

Finally, don't make it easy for bots to perform this operation without the correct identifier. Depending on your requirements, you'll want to avoid (or at least be aware of) situations such as:

Your app treats requests to "/login" and "/%20login" the same
This could break any number of tools aside from bot management, like the logging of more detailed information about sensitive operations that key off the URL path. Part of your DevSecOps practice should be automated functional tests that attempt slightly malformed requests to see how the app responds.

Requests to "/search" are parsed by your app as if they were made to "/login" because of some request parameter value
This is an even more challenging version of the previous example that should be tested for because it can really obfuscate what a client is doing.

A mix of sensitive and nonsensitive operations are made in a single request
This is okay as long as it's still easy to identify that a request contains at least one sensitive operation, or as long as this mix of operations happens after using login as a choke point (more on this in guideline 7)

6. Keep your workflows in the same cookie space where possible

A critical security and privacy feature of all modern browsers prevents them from sending cookies between two different apex domains. Many bot mitigation solutions rely on identifying human/bot behavior as the client makes requests throughout the user journey. This session is usually tracked by a cookie and changing the cookie space during the workflow will complicate how that session information is tracked or how protections are deployed.

Special efforts may need to be taken in situations in which a third-party service must be directly used as part of the workflow, as in the case of a payment processing service where users submit payment information directly to that service's URL, or a cloud authentication service where users are redirected to a login before being returned to the main site (such as with login.microsoftonline.com). Sometimes, those third-party services will take responsibility for protecting themselves against automated abuse, but often they will hold you responsible by charging fees or by threatening to stop doing business with you if you don't protect against bots that come in through workflows on your site. If you have the responsibility to stop bots in these situations, you need to proxy the connections back to the service so that you can stay within your cookie space and set up security controls at that proxy layer. This proxying is best done through the use of a CDN; Akamai uses this exact technique to protect many of its customers.

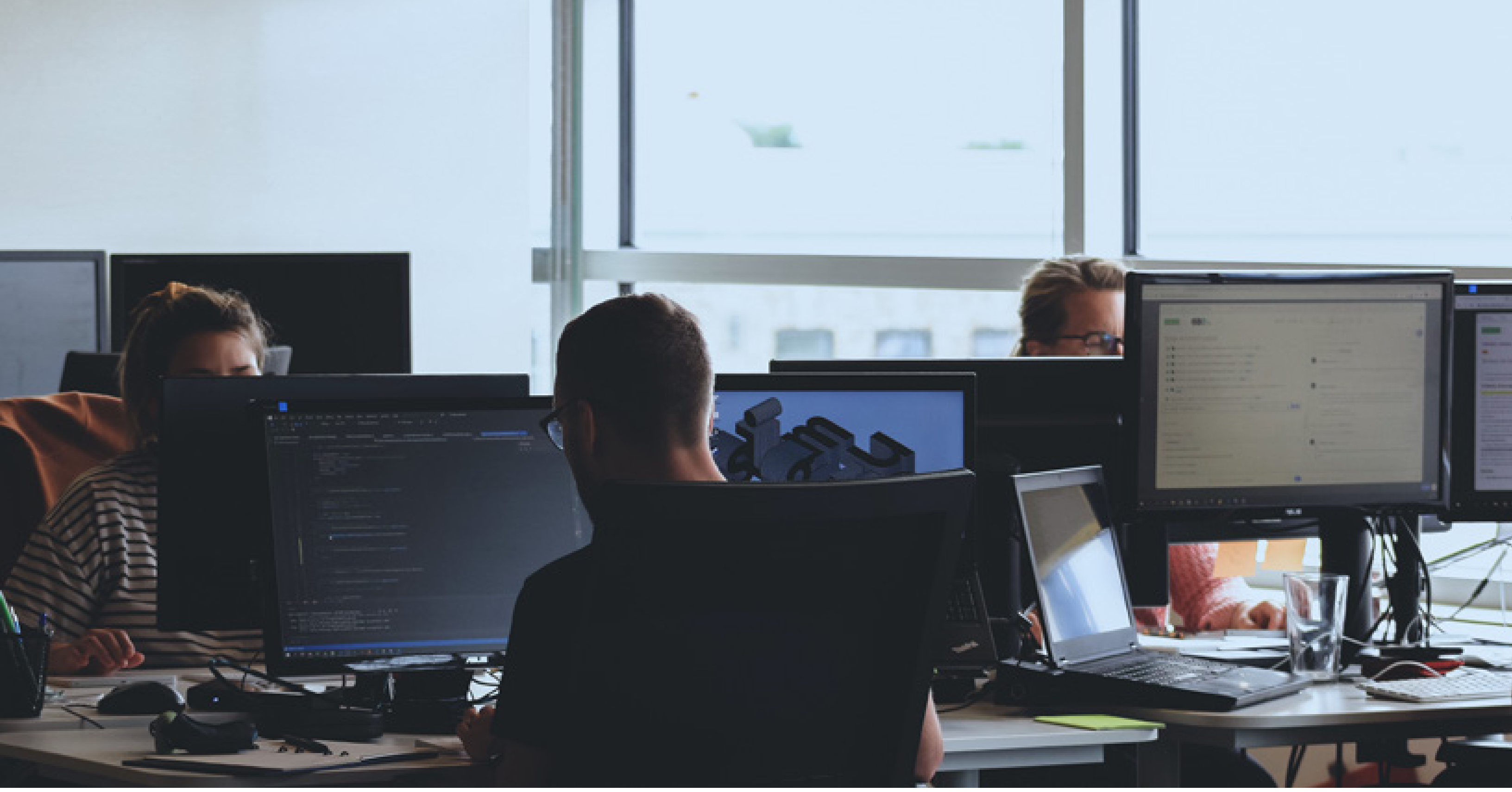
7. Reduce or eliminate sensitive operations that happen before login or other choke points

It would be foolish for a bank to store its money in the lobby rather than its vault, and it would be even more foolish for them to store the money on the sidewalk outside the bank — this is inherently insecure and no number of cameras or guards would make the money as secure as a locked bank vault. To get your cash from the bank, you must first walk through the lobby doors without looking like you're there to rob the place and, once you've done that, you then identify yourself at the counter, usually with a debit card and PIN. If anything suspicious happens in that time, the bank can use its defenses as appropriate — but once you've gone through that process you can make any number of withdrawals or transfers from your accounts as you'd like.

This is how organizations need to think about sensitive operations — before a user is allowed to make a request for a sensitive operation, they should ideally be forced to authenticate (or at least have to click through a page, type in a search term, or otherwise interact with the site somehow). That process is usually sufficient for anti-bot defenses to pick up on the behavior of the user who is interacting with the client and make a decision about whether to allow the sensitive operation to happen. In other words, as the user interacts with a login form, before the login credentials are actually sent back to origin to be validated, anti-bot detections can make a decision about whether to allow that credential validation operation to happen at all. By turning one sensitive operation into a choke point, behind which you move other sensitive operations, you greatly reduce your attack surface and SecOps can focus their attention on defending just that choke point operation.

Login operations are particularly good choke points, both because you can layer in additional security (namely multi-factor authentication, which itself is an excellent tool against both bot-automated and manual credential stuffing attacks), and because you can enforce limits on sensitive operations per account. If only a logged-in user can perform a particular sensitive operation, you can set up a quota or rate-limiting scheme that prevents a single logged-in user from slamming your origin with such requests, and SecOps can review/disable accounts that make an unusually large number of requests for any reason.

Business requirements may complicate this choke point scenario. Take, for instance, a site that shows real-time inventory and dynamic pricing on its product pages: If the business wants this information displayed immediately before any user interaction on the site (such as for users coming in directly from a search engine or bookmark, rather than by clicking or searching from your home page), this is an inherently insecure design and you should reconsider your approach to satisfying the business requirements, or else work with your business partners to modify the requirements. In this case, rather than show real-time data immediately prior to login, perhaps you show *near*-real-time cacheable data until users log in, or at least interact with the site more, after which they can receive the real-time data. Putting the real goods behind login may be totally out of your hands, as businesses tend to push for increasingly frictionless user experiences. In these cases, you must emphasize to the businesses that they are creating extremely alluring incentives for attackers, and they must take more strategic action, beyond mere technological controls, to reign in those dangerous incentives.



8. Reduce the frequency of sensitive operations

If a real user only needs to perform a sensitive operation once per user session, it becomes easier to identify bots that come in at higher volumes. This is typical for logins: Users usually only need to enter a username and password once per session, but a bot operator may be going through lists of thousands of credentials. Even when a user must enter their credentials repeatedly, say in the case of a mistyped password, there will still be user interaction between each login operation.

Ultimately, your goal should be to require some user interaction before a sensitive operation is performed — the more interaction, the better, to give more data to anti-bot defenses.

As an example, imagine a search page that doesn't require authentication and that allows you to set the number of results returned — to 50, 100, or 200 results. An inferior way of performing this search would be for the client to make a request for each block of 50 results; that is, if the user wants to see 200 results on the page, the app will quickly make four successive XHR requests to get the data it needs to build the results page, without any user interaction between each request. A superior way would be for the client to make a single request with the desired number of results specified within that request. This way is better because it would require some user interaction between each operation (e.g., typing in search terms or clicking "next" to see a new page of results), allowing more time and data for anti-bot detections to better determine if a client is being operated by a human even though there could be many search operations performed during a single session of a real human.

To be clear, the more time and user interactions that happen between sensitive operations, the easier it will be to distinguish the bots from the humans — having minimal user input between sensitive operations may not be enough. For example, imagine a search page that brings up results with each keystroke: In an attempt to autocomplete, the page will helpfully make query after query with the most limited of user input. Although there is some user interaction between each sensitive operation (search query), it will be extremely minimal and this makes it more difficult for SecOps when they try to implement security controls. The design should be such that autocomplete from user input is a different function than actually querying for results, just as Google does on their search (you don't see them showing a results page after each keystroke). This way, you can focus on making the autocomplete function more efficient and user-friendly while keeping separate, and reducing the number of, the costlier results query operations. This isn't to say that the autocomplete function can't also be targeted by bots if there is some valuable data in there or if they just want to overwhelm your system, but being able to distinguish between the autocomplete and the search operations will nonetheless help SecOps better defend each one.

9. Authenticate on a separate in-browser page

This is particularly important for clients that run on dedicated streaming devices. You've likely experienced this many times yourself: When you want to log in to a streaming service, you're prompted to get on your phone, visit a URL, and enter a given code on your TV. Although it is more convenient to enter your streaming service password on your phone using a screen keyboard or saved credentials, there is more to that process than most people realize. By forcing users through a login page on their phone, security operators have the chance to gather additional user interaction data — like touchscreen, compass, and gyro events — and make better determinations about whether the device is being operated by a real human being.

Along with clients running on dedicated streaming appliances, it is also good practice to break out the authentication pages for NMAs and SPAs, forcing users to authenticate on a web page before passing an auth token back to the app itself. Doing this allows for more flexibility when it comes to deploying code to protect login, and avoids the need to do larger deployments of the app's entire codebase.

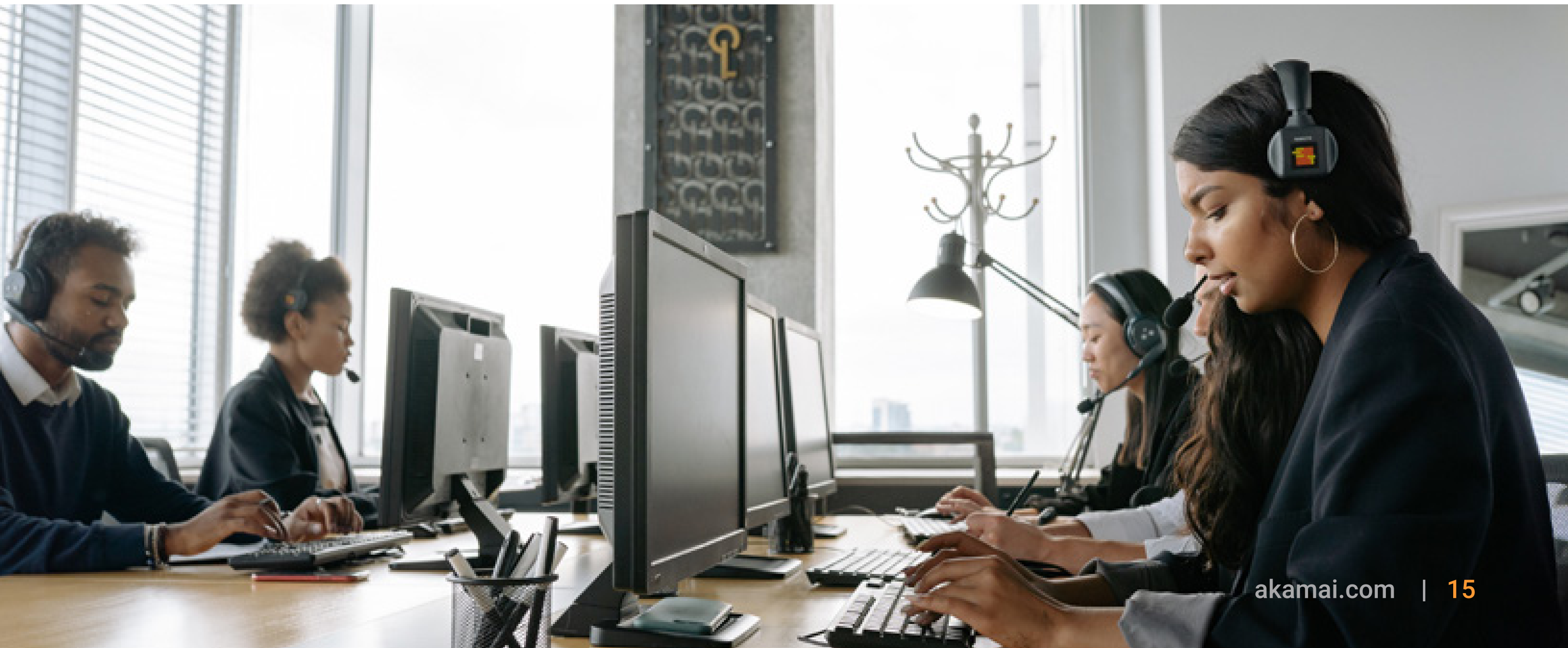
Note: To implement this practice with an NMA, it may be better to launch a distinct web browser application instance, rather than use built-in browser frameworks (OkHttp on Android or WKWebView on iOS), since such frameworks may lack (or disable by default) certain functionality needed by security operators when implementing anti-bot protections. This is not a hard-and-fast rule, but using browser frameworks can introduce additional complexity that could be avoided by launching a browser. Business requirements, however, may dictate that loss of focus for your application is unacceptable, in which case browser frameworks should still be used. If you must use a browser framework, make sure that it completely enables the running of JavaScript and fetches all resources to render the login page from the web server at runtime, rather than render resources packaged with the app. Again, the idea here is to improve the flexibility of developers and security operators so they can implement future security controls specific to the login without having to do a larger deployment.

10. Be mindful of failed login responses and account validation

This is important both to help SecOps identify potential attacks and to avoid leaking information that can be useful to attackers. The response for a failed login that happens as a result of an incorrect password should be identical in its headers and its body to a failed login that happens as a result of an incorrect/nonexistent username — an ambiguous failure message should cover all cases. For scenarios in which accounts are locked as a result of too many failed login attempts, only users who present valid credentials should be informed that a lockout has happened.

It is also best practice to have at least one clear, consistent, documented difference between the response to a failed login and the response to a successful login. The most obvious way of indicating this is to use an HTTP response code, such as 401 (Unauthorized) for a failed login and 302 (Redirect) for a successful login. Response headers can also be used to differentiate between successful and failed logins, but care should be taken to not bury the difference. Different ranges of Content-Length or a particular cookie, for example, do not make it easy for SecOps to understand when there has been an uptick in failed logins just by looking at HTTP logs. If the only consistent difference between a failed and a successful login response is something within the body of the response, it will be nearly impossible for SecOps to differentiate them. This should be avoided entirely.

For account creation and password reset operations, care should also be taken not to reveal whether an account exists at all. Like logins and password resets, design these operations so that responses to failed requests for existing accounts are identical to requests for nonexistent accounts, at least until the client has somehow authenticated themselves, such as by clicking an automatically generated link emailed to the user.



Conclusion

Doing everything we recommend in this paper will not make your app completely impervious to bot attacks, and you should still work with your security team to proactively implement a dedicated bot management solution. But, by following these guidelines, you'll be able to enjoy your weekend, while SecOps deals with another blip on their radar that's not your problem — thanks to your proactive, holistic approach to bot-proofing your apps.

Credits

Editorial and writing
Jacob Lovell

Review and subject matter contribution/ marketing and publishing
Danny Harris
Mike Elissen
David Sénécal
Carley Thornell

Further reading

Understanding Your Credential Stuffing Attack Surface

Configuring Akamai Bot Manager to detect and stop adversarial bots

Top 10 Considerations for Bot Management

Contact Akamai for a deeper dive into which solutions would benefit your organization the most.

Akamai protects your customer experience, workforce, systems, and data by helping to embed security into everything you create — anywhere you build it and everywhere you deliver it. Our platform's visibility into global threats helps us adapt and evolve your security posture — to enable Zero Trust, stop ransomware, secure apps and APIs, or fight off DDoS attacks — giving you the confidence to continually innovate, expand, and transform what's possible. Learn more about Akamai's cloud computing, security, and content delivery solutions at akamai.com and akamai.com/blog, or follow Akamai Technologies on X, formerly known as Twitter, and LinkedIn. Published 11/23.