The Ad Block Battle Theater

A technology white paper for Webmasters and Web developers

by David Levine, Adtoniq CTO October 2017

Background & History

A technology arms race is developing between two well funded armies: Ad blockers vs. Publishers. Ad blockers should be more accurately thought of as generic "blockers" because they block more than just ads, and they represent a significant and growing subset of Internet users who wish to block most advertising technology in order to improve their online performance, privacy, and end user experience. Publishers wish to freely offer their content in exchange for monetizing that content using advertising as an alternative to paid subscriptions. And herein lies the battlefield.

Publishers started the war by chasing advertising revenue to the detriment of their end user experience. End users responded by inventing ad blockers to improve their experience, thus breaking the implicit agreement between publishers and their audience which used to be: Publishers provide their content for free, and in exchange their audience will see advertising to cover the costs of providing that content. Now that implicit contract must be renegotiated, using a combination of technical counter defenses and an open dialog between publishers and their audience.

This white paper describes the two armies battling it out in the ad block battle theater, their technical strengths and weaknesses, and prospects for the future. At stake is the future of the Internet as we know it. We'll start with an analysis of the ad block army, then move on to the publisher army, and finish with a prediction for the future.

The Ad Block Army

The goal of the Ad Block Army is to dramatically improve their online experience. Although ad blocking started as a desktop/laptop browser extension, that threat is broadening with Apple's support of content filters in iOS, new browsers with ad blocking baked in, and firewall vendors and cell phone carriers jumping on board to block ads at the network level. New ad block solutions emerge monthly, requiring publishers to stay abreast of the latest threats and potentially make technical changes to their content management systems.

Ad blockers utilize a constantly evolving, community-maintained filter list to determine which content to block. The default filter list used by AdBlock Plus, the largest ad blocker, is called EasyList, and it is maintained by a dedicated team of contributors as open source using the Mercurial SCM. As of this writing, it contains 56,286 rules. It is updated daily – often several times an hour – as new ads and ad technology emerge.

Ironically, Google and other very large publishers are among the largest financial contributors to supporting the development of the filter list, by paying fees (often thought of as extortion from the publisher's perspective) in order to be on the white-list of acceptable ads.

Ad blockers use only two techniques

Despite the plethora of ad blockers emerging, most ad blockers employ only two basic techniques to achieve their goals, and a thorough understanding of these two techniques and their counter defenses is necessary to overcome the harm done by ad blockers. As the battlefield evolves, more techniques will be developed requiring new counter defenses – and these will be described at the end of this paper – but in the mean time the majority of the ad block threat can be met by understanding how to cope with these two techniques.

Technique #1: Block by URL

Ad blockers prevent browsers from loading out-of-line content by using filter lists consisting of URLs often augmented with standard regex capabilities. These filter lists then leverage hooks in the underlying client software to prevent the network connection from being processed.

For this to work, the browser or other client must provide an extension mechanism that supports the ability to hook or block a TCP/IP network connection. The popular Gecko engine defines an XPCOM interface called nslContentPolicy which is used to implement a content policy mechanism that controls the loading of various types of out-of-line content, and the processing of certain types of inline content as well. Gecko-based ad blockers leverage the JavaScript language binding to XPCOM to implement an nslContentPolicy that provides the core ad blocking logic, and this logic is bundled and installed with the actual browser extension.

Browser extensions are not the only place in the network ecosystem where network connections are thwarted. Ad blocking DNS servers block network connections by augmenting the Resolve DNS function. If the domain requested is on the filter list, a failure is returned, and otherwise a standard recursive resolver is used to retrieve the requested DNS entry from a root DNS server. This supports ad blocking for all types of devices (mobile, desktop, etc.) and all types of software including native mobile applications. What is particularly interesting about this approach is that it does not require any software to be installed by the end user and in fact the end user behind a router using an ad blocking DNS server has no way to disable ad blocking, short of changing network connections.

Firewalls are also an easy place to implement network-based ad blocking, from the firewall built into the cheapest home router to professional grade enterprise firewalls for large corporate networks. All one needs to do is copy and paste one of many publicly available filter lists into a “block” firewall rule, and the network connections will be blocked for all users behind the firewall. Not all firewalls support regular expressions in firewall rules, but that doesn't prevent most ads from being blocked.

When blocking a network connection, ad blockers can do this in one of two fundamentally different ways, leading to two very different ad campaign metrics, and it's important to understand which kind of ad blocking is happening in order to understand how accurate publisher reporting is. Misunderstanding this can lead to a publisher billing an advertiser for impressions that were never seen by ad blocked users.

If the ad blocker prevents the network connection from being opened, the server (for example an ad server) will never even know that a user was there. On the other hand, the ad blocker can open the network connection, stream the bits from the server but drop them on the floor (/dev/null) so that they are never seen by the user. This would mislead the server into believing that it had responded to a request or served an impression.

Technique #2: Block by CSS Selector

While URL rules can block entire services from operating such as ad servers, analytic systems, and multivariate testing tools, often times advertisements are site-served or otherwise embedded in HTML content that is not blocked by a URL. In these cases, the advertising content is embedded inside other content that the end user wants to see, which leads to the requirement that the ad blocker must be able to target nested content. CSS selectors are typically used by ad blockers to identify nested HTML content that should be blocked. If the content is not ultimately rendered as HTML, CSS selectors can't remove it.

Typically CSS selectors are used to select DOM elements with DOM classes or ids matching values that often indicate ads, such as “adchoices” or “leaderboard”. However many more sophisticated CSS selectors are used as well looking for combinations of attributes, or child elements within specified parent elements.

There are some limitations with this scheme. First, CSS3 Selectors have no mechanism to target the parent of a matching node. The W3C specification for Selectors Level 4 is still in draft form but will eventually support this capability using an expression like

```
div > !.myClass > span
```

Such an expression would allow blocking of the ad unit containing an element of a given class. However browser support for CSS4 is still spotty at the moment. And even when implemented, CSS4 still will not support the ability to select a node according to text within it. While this could be implemented in JavaScript (and is implemented as `:contains` within JQuery), it is not part of the CSS4 specification which is what powers (and limits) the popular ad blockers.

Another issue when blocking by CSS Selector is that the selector can not be too generic, otherwise it would generate an unacceptable number of false positives. This is a significant issue for ad blockers, and is an area of active discussion in the community as new filters emerge daily that occasionally have unintended consequences of blocking site content or functionality that end users desire. When false positives are detected, the rules must be either fixed or removed altogether.

Identifying the content to be removed is the first step. Once identified, the content must be removed from the experience. In many cases, the ad blocker can not simply delete all the DOM elements matching the CSS selector because that would break the site. Instead, they generally hide elements by setting various CSS attributes including `display` and `visibility`, and the element attribute `disabled`. This is important, as we'll see later, because counter defenses can recover and leverage these hidden elements for various purposes.

Going beyond CSS Selectors

Recently as publishers found ways to evade CSS Selectors, the ad block army responded by adding additional pattern matching capabilities. For example uBlock added the ability to target inline scripts containing specific content, for example with a rule like this:

```
##script:contains(“isAdBlockedUser”)
```

The main limitation of this approach is that only Firefox contains the necessary platform features required to implement this, so this is a FireFox-only solution at the moment. Even worse, because of this limitation, not all ad blockers find it worthwhile to support this feature.

The “stealth” ad blocker ideal

Ad blockers would ideally like to operate “stealthily”, meaning that the publisher can not detect that the user is using an ad blocker. If ad blockers can achieve this, then the publisher will not attempt to protect any content and will treat the user as if they had no ad blocker, even though ads are being blocked.

Unfortunately, there is no way in principle that ad blockers can achieve this goal using today's Internet. That may change in the future (more on that later), but for now, they can not operate without being detected.

However, because many publishers fail to implement robust ad block solutions, ad blockers can effectively operate undetected by some web sites. These publishers are unaware of the extent of their ad block threat, and likely don't know that their ad block analytics are inaccurate.

The Publisher Army

The goal of the Publisher army is to monetize their ad blocked users. Publishers have a range of options available to them to achieve this:

1. Thwart the intentions of the ad blocker.

2. Block valuable content from being viewed by ad blocked users.

3. Engage ad blocked users in dialogue to explicitly renegotiate the broken contract between publisher and viewer.

Different publishers take different approaches based on their unique audience profile. But it all starts with detecting whether or not a user's experience is being filtered by an ad blocker either on their device or somewhere in the network fabric.

Detecting ad blockers

To detect ad blockers, publishers embed a small JavaScript in their page that uses a bait and trap pattern. The code dynamically lays bait by creating an invisible ad unit somewhere on the page after it loads, using elements that will match a CSS Selector rule on the default filter list of ad blockers. It then enters a poll loop, periodically checking to see whether the ad blocker has taken the bait by hiding the ad unit. If so, this indicates the page has been affected by an ad blocker.

This technique does not work for router or firewall based ad blockers because they can't remove ads that are added after the page loads. This is a limitation of network based ad blockers because this opens the door to create ads dynamically, after the page loads. But it also means a different detection technique is required.

Detecting network-based ad blockers requires that the publisher include out-of-line content that will match a URL rule on the default filter list of ad blockers. This out-of-line content includes a JavaScript which sets an internal flag indicating that it has successfully loaded, which proves that it was not blocked by a URL pattern on the filter list.

Meanwhile an inline script in the publisher page is in a pool/wait loop waiting for the flag to be set. If the flag is not set within a reasonable amount of time, the code concludes that the ad must have been blocked by a URL rule in the filter list. While this technique for detecting network-based ad blockers works, it can be slower because of the time delay required to prevent false positives – basically how long the page will wait for the out-of-line content to load. Because of this, the bait and trap technique is used in parallel because it will usually trigger first.

If the code triggers a URL type of rule on the filter list while at the same time does not trigger a CSS Selector rule, then the code can reasonably conclude that the ad blocking occurred in the network fabric and not within the browser or other client.

It's important to know where the ad blocking is occurring in order to provide appropriate messaging to your ad blocked audience. If ad blocking is occurring in the network fabric, you can't ask the user to disable their ad blocker because they don't have one. However you can give this choice to browser-based ad blockers.

Once the code detects that a user has been affected by an ad blocker, it needs to communicate this information to an analytics system somewhere. The next section covers how this is accomplished in light of the fact that commonly used analytics systems are often blocked by ad blockers.

The “stealth” ad block detection ideal

Just as ad blockers wish to operate stealthily, publishers wish their ad block detection code to operate without being detected. If the ad blocker can detect the publisher's script, it can try to disable it. So remaining undetected is vital.

Keeping the script inline rather than out-of-line is the first step to being undetected. This is important because it is trivial to write a filter rule to block the URL for the out-of-line script.

Next the inline script should be combined with other functions that are absolutely essential to the functioning of the site. This is important because this makes it impossible in principle for an ad blocker to remove any single inline script from the publisher's page without breaking the entire site, which would lead to unacceptable results for the ad block community.

Lastly, the script must be scrambled using polymorphic script encryption as described next.

Polymorphic script encryption

Polymorphic script encryption scrambles the ad block detection script differently every time it is embedded into the publisher's page, effectively making it impossible to detect. This borrows a page from polymorphic virus technology, allowing the script to transmute itself into infinite variations. In addition, the script can include check digits to ensure that an ad blocker has not tampered with the code.

This is an important counter-defense to evade the uBlock feature for FireFox which can detect inline scripts by searching for text in the script.

Ad block analytics

Increasingly publishers are tracking new Key Performance Indicators (KPIs) to measure the extent of their ad block problem including:

1. Percent of ad blocked page views and users
2. Advertising revenue lost to ad blockers

Publishers use these KPIs to drive both technical and business process changes to improve these metrics. Ideally publishers build multivariate testing on top of these KPIs to further speed up the optimization.

While publishers should be looking at these new KPIs, it's critical to understand the effect that any ad block strategy can have on other KPIs. For example, if publishers take a hard core stance and require their audience to disable their ad blocker to view the site, the publisher will dramatically reduce their percent of ad blocked page views for sure, but they will have fewer overall page views because some percent of their audience will not accept that deal. That could either increase or decrease total ad revenue, and publishers need to understand which is happening and why and be prepared to react quickly.

In order to gather these metrics, a JavaScript must execute within the browser to AJAX the analytics data to an analytics collector that will record whether the user is affected by an ad blocker. Many ad block solutions on the market leverage leading analytics solutions such as Google Analytics or Omniture to capture this information, but if the ad blocker is blocking network connections to those servers (as is often the case), then the publisher will not receive any data regarding these ad blocked users.

To reliably AJAX analytics data from the browser, the script must send the data to a URL that can not be on the ad blocker's filter list. There are only two ways to achieve this:

1. Implement inverse site serving and leverage the fact that the publisher's own web site can't be on the filter list, and send the data back to the publisher's own web site where it must be processed and recorded, or;

2. Send the data to an analytics collector at a URL that can't possibly be on an ad blocker's filter list.

Solution #1 turns the publisher's content management system into an analytics collector. CMS's are generally not designed to do this so this approach is not recommended. Solution #2 is covered in the next section.

Generating throw-away random domains

Solution #2 can be implemented by frequently generating new, random domains to front an analytics collector. If the random domains are automatically generated faster than the manual army of filter list maintainers can add them to the filter lists, then the domains will effectively never be blocked. The main advantage of this approach is that it works with a publisher's existing server infrastructure with little integration overhead. The main limitation is that this only works for participating or cooperating analytics systems that are designed to work with the random URL approach.

Because this approach can only work with cooperating analytics systems, it can't leverage third-party data of non-cooperating systems such as is available in Google Analytics or Omniture to create demographic profiles of the publisher's ad blocked audience. However that limitation can be ameliorated so long as the third-party data publishers can collect is statistically significant and a sufficiently representative sample, but more about demographics later.

In order for the random URL domain to work, those random domains must be communicated from the analytics collector directly to the publisher's CMS web servers, in a timely and secure way, and then immediately used within the pages that are served to users. Once communicated, the CMS then uses the random domain until a new one arrives.

Random domains can be applied beyond the use case of collecting analytics data, to any other server in the stack. For example, a cooperating ad server fronted by a random domain could serve any kind of an ad in an iFrame, and bypass the URL filter list of ad blockers. In order for that ad server to be 100% invisible to ad blockers, it too must recursively utilize random URLs for all content it loads, such as its own images, JavaScript, or other resources loaded as part of the ad.

In addition, because ad blockers use regular expressions to augment URL pattern matching, the publisher must also be careful that no part of the URL following the FQDN can be used in a filter rule.

Hiding in the CDN

Ad blockers generally have problems blocking content in a publisher's CDN because that contains the publisher's editorial content and not their advertising content. They can't block the whole CDN domain as that would break the site. The publisher can exploit this by embedding their advertising content in their own CDN, effectively site-serving their ads. While this technique does not work for older generation ad technology, newer ad technology is emerging that can be configured to write their content into the publisher's CDN.

This alone is not sufficient, because any individual URL can simply be added to the filter list, even if it is on the publisher's CDN. To work around this, the advertising content must be re-written into constantly changing URLs that are indistinguishable from normal editorial content. That is, the URL can not be of the form

```
//mycdn.com/adcontent/random-
stuff
```

because ad blockers could easily target the “adcontent” part of the URL.

Bypassing CSS Selectors

Ad blockers use CSS selectors to hide content embedded within the publisher's site. Publishers can bypass CSS selectors by randomizing the various styles and attributes that can be targeted by CSS selectors so that they can never get a solid match. Unfortunately, most web sites are not designed with this in mind, so it is usually preferable for a publisher to build or buy a new subsystem that automates this.

To implement this, new server-side logic is required to generate random class names and ids for DOM elements, and then use those names and ids in all the appropriate places such as within JavaScript or CSS classes. Newer ad servers are designed with this in mind, to prevent content from being easily blocked.

In addition, the values of other attributes need to be adjusted as needed. For example, some filter rules look for specific custom attribute names or attribute values, or children of other elements, or they can even target popular ad unit element dimensions such as the 300 x 250 ad unit.

As new filter list rules emerge, the randomizing code must be upgraded to bypass the latest threats. This is an area where publishers may be better served by outsourcing this maintenance to a vendor that can amortize the cost of maintaining this randomizing code across many publishers, rather than building it in-house.

Ad block demographics

The demographics of a publisher's ad blocked audience are likely to be different from their overall demographics. For example, they are likely to be skewed towards younger, male, more technical users. Publishers can leverage a detailed knowledge of their ad blocked demographics to design messaging for their audience that will optimize the chances of a successful outcome.

Demographics can only be collected if first-party or third-party data is available about users. If a publisher has first-party data available, that data can be sent to the analytics system to create a demographic profile. However, many publishers must rely on third-party data to create a demographic profile. Because some ad blockers block third-party analytics systems and some do not, the demographic data available about ad blocked users will represent only a subset of the ad blocked users. Nevertheless, if a sufficient number of users do not block third-party analytics then sufficient data may be available to draw meaningful conclusions about the ad blocked audience demographics.

Dynamic Ad Block Strategies

Publishers can optimize their revenue recovery from ad blocked users by dynamically generating different ad block strategies based on targeting signals available about the user. For example, if the referring URL is from a social media site like Facebook, a more lax strategy can be used initially.

Or if the same user has been seen with an ad blocker multiple times, the messaging presented to the user can be increasingly more urgent from page view to page view. This approach requires some way to track the user over multiple visits, most likely using a cookie. However, increasingly publishers are using fingerprinting techniques unrelated to cookies.

Hiding content from ad blocked users

Publishers can choose to protect some of their content from ad blocked users by essentially turning the tables on the ad blocker, and using the very same techniques that the ad blockers use, only on their own high value content. That is, publishers can use CSS Selectors to select their own high value content to hide, and then either delete or hide those DOM elements.

The simplest and least disruptive approach for publishers to implement this is to use an embedded script to hide or reveal the protected DOM elements after they are loaded in the browser, based on whether an ad blocker has been detected. This is called client-side protection because the protection happens on the client side, after the content has already been sent by the server. While this approach works for the average user, the content is still readily accessible to the more technical person who looks at the source to the page. Nevertheless this may be sufficient protection in many cases, because the user could just as easily get the content by pausing their ad blocker and refreshing the page, which is probably easier than viewing the source to the page.

Server-side protection

A more robust solution is called server-side protection and it works by having the CMS detect that the user is affected by an ad blocker and not ever sending the protected content to the user. The challenge with this approach is that when the HTTP Get or Post request arrives from a user, the CMS does not yet know whether the user is affected by an ad blocker, because the ad block detection script has not yet run on the client. It is not sufficient to rely on the results of the previous page's ad block detection because the user can easily enable their ad blocker between page views; some well known sites contain this vulnerability as of the time of this writing. Also if this is the first page view of a session, there will be no previous page view to determine the user's ad block status. Consequently, the CMS needs to send the page back to the user without knowing whether they've got an ad blocker.

To solve this problem, the publisher can utilize iFrames or AJAX calls to asynchronously load content that should only be displayed to non-ad blocked users. The ad block detection code dynamically loads the iFrame or calls the AJAX if and only if no ad blocker is detected. To guard against replay attacks, the iFrame/AJAX call can pass in a server-generated nonce which is embedded into the page and sent back to the server for verification prior to sending the content back.

Unfortunately most web sites are not designed to load their content in this way, and they must be retrofitted to some extent to support this approach if publishers wish to support server-side content protection. Using CMS plugins designed to support server-side protection can greatly simplify the adoption of this strategy for publishers.

To be clear, the server-side protection approaches described here are not intended to replace proper digital rights management (DRM), which supports cryptographically secure content protection.

Providing teaser content

Rather than taking an all or nothing approach with respect to hiding individual DOM elements from ad blocked users (hide the entire element or show it), the publisher can choose to present some of the protected content as a tease with an offer like “disable your ad blocker to reveal more.” This can take several forms including: blurring content, dimming content, or only showing partial content.

When using client side protection, JavaScript code can apply the necessary transformations to DOM elements to achieve the desired teasing effect. With server-side protection, two versions of the assets must be prepared: the tease version and the full version. To fully protect these assets, they should be accessed either through a server-generated nonce, or a frequently changing random URL.

This presents a special challenge when streaming content off commercial CDNs because they are not able to validate a nonce or be fronted by a random URL. If a publisher is concerned about protecting their CDN content from direct access, the only approach that can work and still utilize the CDN is to regularly re-write the CDN content to new, randomly generated URLs to prevent them from being replayed out of context. At the same time, the publisher needs to remove out of date URLs so that they are no longer available.

Video is a special case because it provides an excellent opportunity to stream the first part of the video to ad blocked users, getting them hooked, and just at the "right" moment, publishers can force the video to pause and then show a message like "to see this rest of this video, please disable your ad blocker." This is implemented by creating two separate video streams: One for ad blocked users, and the full one for non-ad blocked users.

Interacting with ad blocked users

As publishers started interacting with their ad blocked users, the ad block community responding by creating the "anti-adblock filters" list, which blocks the messages publishers use to communicate with their ad blocked users. As a result, when interacting with ad blocked users, the DOM elements created for that interaction must use the random ID, class name, and other techniques described in the *Bypassing CSS Selectors* section.

When interacting with ad blocked users, it is vital to understand what kind of ad blocker they are using. For example, if they are using a network-based ad blocker, the publisher should not ask that user to disable their ad blocker because they most likely can't. However if they are using a browser-based ad block extension, the publisher can deploy ad block fingerprinting code to determine which ad blocker the user is using, for example AdBlock Plus or AdBlock. Using this fingerprint, the publisher can offer customized instructions on how to white list their site, describing the precise steps for that ad blocker.

Re-inflating hidden content

When ad blockers find inline content to block, they typically hide the matching DOM elements rather than delete them. This is done because deleting the elements often breaks the web site. These are most often ad units.

Because the DOM elements are hidden and not deleted, they are available to be "re-inflated" which means that they are made visible again. This is easy to do simply by targeting the DOM element and resetting its attributes to be visible.

Once made visible again, it's up to the publisher to decide what do with this space that was just revealed again after being hidden by the ad blocker. They have a space on their web site where they know they can communicate with their ad blocked audience. They can choose to:

1. Serve another ad, so long as that ad will not be blocked.

2. Engage in a dialog with the ad blocked user, motivating them with a call to action to either disable their ad blocker, sign up for a subscription, or take some other action.

The future battlefield

At present, publishers are able to detect all ad block threats, assuming they are using robust ad block detection technology. This will change in the short term as ad blockers add new techniques beyond their currently limited two-weapon arsenal (URLs and CSS Selectors). In the long term, several significant efforts are under way to radically reset the entire world wide web technology landscape with an eye towards properly solving the problems ad blockers only partially solve today. However these long term approaches are so radical at the moment that their future is not at all certain, and in any event is years away, which leaves us with the short term future to worry about.

In the short term, ad blockers must go beyond their two technique approach of blocking network connections by URL or CSS Selector. What they will eventually need to do is add script-based rules, that can execute arbitrary JavaScript when they match a pattern. This is the only way that ad blockers can ever hope to defeat the counter defenses of publishers, and perhaps operate undetected by the publisher.

While this is what is needed technically, this approach is also quite problematic in practice because that JavaScript code is created in an open source environment and must be very carefully tested and controlled to prevent the introduction of malware, which would defeat the very purpose of ad blockers. Browser extensions are notorious vectors for malware, and there is no readily apparent way available to harmonize these opposing forces.

Ad blockers also need to develop more sophisticated pattern matching capabilities that go beyond the capabilities of CSS Selectors, possibly using some of the extended capabilities available in JQuery, or simply using script-based pattern matching.

The short term future for many publishers is mostly about implementing a basic ad block detection and remediation strategy, paying special attention to ensure that ad block analytics are accurate. In 2016 most publishers are evaluating ad block strategies and will make a build vs. buy decision, evaluating a growing range of solutions appearing on the market.

Publishers also need to ensure that it is not trivially easy to thwart their intentions simply by turning the ad blocker off upon entering their site and then turning it back on once they are in.

In the long term as ad blockers evolve, publishers must move more towards a Digital Rights Management (DRM) model, giving them better control over their content, including who gets to see it and under what conditions. DRM is based on cryptography and relies on using a secure lock-box on the client to mediate access to the content.

Such a long term future is not easy to deploy today because the web and content management systems in general were not designed with DRM in mind. However market forces are driving the industry towards new web standards and software that will effectively reintermediate the entire advertising ecosystem and technology stack.

With DRM in place, publishers can configure rights policies that include subscription or micro-payment policies, time-based policies, and ad block status policies, among others.

But the bigger question is: How will publishers and their audience renegotiate the broken contract that used to be free web sites in exchange for advertising? Will this model continue or will another model dominate the Internet? What do you think? Let me know at david@adtoniq.com.