An Analysis of various web tracking methods

William Huba
An Analysis of Various Web Tracking Methods

by

William Huba

Committee Members

Bo Yuan
Yin Pan
Sumita Mishra

Thesis submitted in partial fulfillment of the requirements for the

degree of

Master of Science in Networking and System Administration

Rochester Institute of Technology
B. Thomas Golisano College

of

Computing and Information Sciences

11/16/2012
# Table of Contents

Table of Contents........................................................................................................................................... 2

1. Abstract.......................................................................................................................................................... 5

2. Introduction ................................................................................................................................................ 5

3. Related Work ................................................................................................................................................ 6

4. Methods of profile detection .......................................................................................................................... 9

   4.1. Description of storage ............................................................................................................................... 9

       4.1.1. Etags ................................................................................................................................................... 9

       4.1.2. Flash Longterm Storage Objects .................................................................................................. 11

       4.1.3. JavaScript Localstorage ............................................................................................................... 13

   4.2. Geolocation ............................................................................................................................................ 13

   4.3. Fake DNS ............................................................................................................................................... 14

   4.4. Fingerprinting ....................................................................................................................................... 16

       4.4.1. Browser matching ......................................................................................................................... 16

       4.4.2. OS fingerprinting ......................................................................................................................... 17

       4.4.3. System Fonts ............................................................................................................................... 19

       4.4.4. Plugins ........................................................................................................................................ 21

5. Profile algorithm .......................................................................................................................................... 22

6. Test results ................................................................................................................................................... 25

   6.1. Browser behavior .................................................................................................................................. 25
6.2. Summary of methods ................................................................. 27
6.3. Existing vendor solution .............................................................. 28
6.4. Defeating the tracking methods ..................................................... 29
7. Conclusions .................................................................................. 31
8. Bibliography ................................................................................. 33
9. Appendix ....................................................................................... 35
  9.1. ActionScript code .................................................................... 35
  9.2. start.py .................................................................................. 37
  9.3. dns_track.py .......................................................................... 39
  9.4. p0f_parse.py .......................................................................... 43
  9.5. Django code ........................................................................... 46
      9.5.1. Root files ........................................................................ 46
          9.5.1.1. settings.py ............................................................... 46
          9.5.1.2. urls.py ................................................................... 51
          9.5.1.3. views.py .................................................................. 52
          9.5.1.4. profiler.py ................................................................. 56
      9.5.2. templates directory .............................................................. 68
          9.5.2.1. base.html ................................................................. 68
          9.5.2.2. meta.html ................................................................. 72
          9.5.2.3. allprofiles.html ......................................................... 72
9.5.3. profiles directory ...........................................................................................................73

9.5.3.1. models.py ..................................................................................................................73

9.5.4. static directory .............................................................................................................76

9.5.4.1. flashiso.js ..................................................................................................................76

9.5.4.2. plugindetect.js ...........................................................................................................77

9.5.4.3. style.css .....................................................................................................................77
1. Abstract

The accurate tracking of web clients has historically been a difficult problem. Accurate tracking can be used to monitor the activity of attackers which would otherwise be anonymous. Since HTTP is a stateless protocol, there is no built-in method for tracking clients. Many methods have been developed for this purpose; however they primarily rely on the cooperation of the client and are limited to the current session and are not designed to track a client long-term or through different environments. This paper takes an in-depth look at the most popular methods of tracking web users and how well they preserve information when a client attempts to remove them. Each method is evaluated based on the amount of unique information they provide and how easy a client can defeat the method. The tracking methods are then combined using a profiling algorithm to correlate all of the available information into a single profile. The algorithm is designed with different weights for each method, allowing for environmental flexibility. Test results demonstrate that this approach accurately determines the correct profile for a client in situations where the individual methods alone could not.

2. Introduction

Most methods of tracking web attackers are not perfect and can be defeated very easily once the attacker is aware of them. Currently, the majority of tracking tools look at IP addresses in at least some capacity, and in many cases primarily rely upon it; hence a smart attacker would bypass IP tracking through spoofing or proxies, a relatively simple thing to do. A tracking method gaining recent popularity is storing identifying information on the web client. Client storage provides almost no chance of a false positive when done properly; however, it is very easy to generate false negatives. It is technologically feasible for an attacker to prevent any kind of storage based tracking by simply blocking the storage objects or deleting them. Using the privacy mode built into the Firefox browser, or any similar means of
sandboxing the web browser, causes the client storage to be lost when the browser is closed. With VMs, reverting to a snapshot would cause the same effect. These simple workarounds prevent current tracking methods from being useful against a person that expects they are being tracked.

The current effectiveness of using exclusively client side storage is mostly based on the fact that it isn’t a very popular technique and preventative measures are still being developed. Kerckhoffs’ Principle observes that obscurity is an ineffective approach to security and a truly secure system must assume that the attacker knows how it operates [1]. Although normally discussed in relation to cryptography, the same requirements can be applied here. Using only a tracking method that relies on being unknown will not for long. In order for a tracking system to be difficult to circumvent a more holistic approach is needed which takes into account additional factors. A robust tracking method should allow an attack to bypass any single method without the entire chain falling apart.

The ability to track attackers over multiple web sessions strengthens the ability of defenders to prevent an intrusion; however, currently used tracking methods are too easily circumvented to be relied on for inter-session correlation. This thesis will present a comparison of various tracking methods and the effect of combining them into a single profiling algorithm.

3. Related Work

An important distinction to make when attempting web based tracking is the separation of profiles from sessions. A session is the set of web requests made by a single user that can be correlated into one activity set while a profile is a correlation of all sessions that are known to be from a single user or entity. Analysis of user sessions has shown that sessions cannot be determined using time cutoffs; however, time analysis can help distinguish between requests coming from people and those coming from automated tools [2]. One of the signs that requests are being automated is that they are coming in
regular intervals, regardless of whether the intervals are long or short. Users are always much more sporadic with their requests.

The most common method for profiling a person visiting a website is local storage on the client. Most websites utilize cookies to this end; the client stores what the server wants it to store and then sends that information back the next time it visits the server. Normal HTTP cookies are very easy to find and delete, or even block. All major browsers support some level of cookie management. However, there are numerous ways to store information more persistently in places that are not normally used for client profiling.

Client side storage beyond simple HTTP cookies has been examined as a way to authenticate users [3]. Although only the browser cache was discussed, the same concept can be easily extended to essentially anything that the browser will store. At its core, client side storage provides a way to identify a particular browser or entire computer in some cases. Although many have viewed this as a privacy issue [4], the technique can be used by defenders to help identify and track web attackers.

As a proof of concept, ‘evercookie’ has implemented 13 different ways to force client-side storage [5]. If any single object remains when the client revisits the website, all of the missing ones can be recreated. Evercookie utilizes the storage of some common user extensions, Microsoft Silverlight and Adobe Flash. These objects are not managed by the browser; in fact, they have their own storage areas completely separated from the browser and can share their objects across multiple browsers on a single computer. Flash cookies, or Local Shared Objects, are unique to each user account but are shared among browsers [6]. The remaining objects, such as settings etags in the image cache, all exploit the behavior of browsers and are only accessible from the browser they were set on.

A separate problem from maintaining accurate profiles is determining whether multiple requests are from a single person. With automated tools, particularly tools used for crawling websites, there is little information in the HTTP requests that allows them to be grouped together. It is less likely
that there is a user agent string sent, as there is in full browsers. Additionally, many tools used for
crawling websites by sending HTTP requests to every linked page on a website do not store cookies. IP
addresses are likely to remain stagnant for a given session; however, this does not give an accurate
depiction of users behind NAT-enabled routers. Any number of requests could potentially come from
any number of people, all behind a single IP address. Other properties of the connection need to be
examined in order to identify these tools consistently.

Tracking the ID field in IP headers can help correlate multiple packets to a single session [7]. The
ID field is incremented by the kernel for every IP packet sent out. Although there will be some gaps in
the numbering for packets going to other servers, in the majority of cases the distribution is close
enough that it can be used for correlation. One potential consequence of this approach is false positives
caused by other clients that happen to have similar numbering in their packets. This possibility prevents
the ID field from becoming an authoritative identifier; however, it is still very useful as a weighted
metric.

Another potential tracking method is to force the attacker to reveal his location through
automated requests involving a unique token. The Decloak project does this through DNS [8]. An image
or link is included on the website that is formatted as a unique identifier subdomain, such as
“UID.domain.com”. The subdomain resolves to the same address as the parent domain, so no
functionality is lost, but the request is logged by the DNS server. By using this method the attacker’s IP
address can be revealed, even if he is using a web proxy. Using unique identifiers can also be used to
track a person that is using a web spider or bookmarked links. In addition to DNS records, there are
several places that unique IDs can be placed to this effect, such as SSL/TLS certificates or otherwise
meaningless CGI attributes [7].

The Decloaking engine is the closest example to the work done here [8]. However, it is different
in two important ways. The first is the design of the engine. There methods implemented in it are based
solely around identifying the IP address of the user. There is no cross-session tracking, despite the fact that many of the techniques would lend themselves well to such an application. Secondly, and perhaps more importantly, the results are not correlated. Each address detection method reports its own findings, but they must be manually examined and any discrepancies are left open for interpretation. In order to effectively and reliably track web attackers many different methods must be used together and not just separately.

4. Methods of profile detection

The tracking methods that were analyzed and implemented can be divided in three categories; client storage, IP detection, and fingerprinting. The reliability against being blocked, resilience to tampering, and ability to provide information unique to the specific client are discussed for each method in these categories.

4.1. Description of storage

The tracking techniques described in this section each rely on the client storing a value sent to it by the server. For every storage method, it is possible for the client to detect the value being set or retrieved and prevent or modify it. The difficulty and likelihood of this happening is different for each storage location, giving them unique reliability characteristics.

4.1.1. Etags

Etags are a standardized method of controlling cached content. They are intended to be used as signatures that validate whether a client's cache is updated, but can be used to store and retrieve arbitrary information for tracking purposes instead. In a normal HTTP flow involving cached content, the web server sets the etag in the HTTP response header after a client requests a resource configured to use caching; the client stores that value along with the returned content in its cache. The next time the
client makes a request for the same resource, it will include the value of the etag as "If-None-Match". If the current etag matches the one sent by the client, a status code of 304 (Not Modified) is returned; otherwise the content is sent with the new etag [9]. Unlike the "Last-Modified" header, the actual content and format of an etag is completely arbitrary. Although most legitimate web servers will either provide a file version or a hash of the represented resource, there is no requirement to do so. This makes etags very useful for storing an identifier and regenerating cookies. Figure 1 below shows an etag being set in the response headers for an embedded image. In the response headers, the Must-Revalidate header informs the browser that it should always check its stored etag value against the current value on the server. The only caveat with using this method to store a unique identifier is that the etag is set for only a specific URL and not the entire domain like a normal HTTP cookie. Browsers will accept and render new content even if the etag has not changed, so there is no harm in sending the same etag for every request during a given session along with the requested content.

Figure 1

This technique has been recently discovered to be in use by several large websites as a way to track users, even after the users delete their cookie. Researchers found that Hulu was tracking users by storing identifying information in both etags and Flash LSOs [10]. This was done using a commercial service called KISSmetrics which other lesser-known sites were using for similar purposes. Despite the practice being exposed, this demonstrates how easy it is to hide and retrieve tracking information from most clients unless each possible medium where it can be stored is intentionally examined.

While this method should work on every HTTP compliant browser, there are differences in how browsers actually reload content which affect whether requests are actually sent to the server for
cached content. Each browser implements caching differently and the specific way that user reloads the page often has an effect on whether the cache is re-validated. As an example, if a user just presses the ‘enter’ key in the URL bar to reload a website Chrome will load static resources from its cache without attempting to validate them. If the user clicks the reload button or reloads with a hotkey instead, then the browser will validate the cache. Even when the browser does send an HTTP request for a given resource, etags are still not completely reliable even in a fully compliant browser. The RFC states that while they may be used for resource comparison, clients are not actually required to add them as an HTTP header [9]. A browser can request and the full resource every time it loads a page and not use etags as a cache control without violating the RFC, but few clients do this as it causes unnecessarily high network utilization.

When the etag value is sent to the tracking server, it is handled similarly to a standard HTTP cookie. Unlike normal cookies, etags are more difficult to fake in standard HTTP clients and can thus be assumed to be more accurate. The value can still be easily removed by erasing the browser’s cache, and a user could block etag tracking completely by disabling caching or using a client that doesn’t support caching. Due to the ease of removal, etags have a lower persistence ability than other methods and cannot be relied upon as a sole storage method.

4.1.2. Flash Longterm Storage Objects

Adobe Flash Player is commonly found on modern systems and is used by websites as a way of creating and showing dynamic content. As of July 2011, some version of Flash can be found on 99% of Internet enabled computers [11]. Despite not being a web standard, it is pervasive enough that it can be reasonably relied upon to be present on a client’s browser and used as a tracking vector.

Flash applications can store any information they want inside of Longterm Storage Objects, or LSOs [6]. Similar to HTTP cookies, LSOs are normally used to save preferences or other specific user
information on the client's device and can be retrieved by the server when the client revisits the same site. However, one critical area where LSOs differ from HTTP cookies is that they are shared across multiple browsers. The cookies are stored as ".SOL" files in their own cache outside the control of the browser [12]. Since Flash is a third-party software package and not managed by the browser, all browsers installed on a single device will use the same location for LSOs. The exception to this is Chrome, which integrates Flash directly instead of having users install it as a plugin [13]. Chrome uses its integrated control of Flash to force LSOs to be stored in a Chrome-specific environment instead of sharing the storage with other browsers.

In addition to being shared across browsers, Flash LSOs previously were not sandboxed when using the various privacy modes offered by browsers. When in an ‘incognito window’ for Chrome or ‘private browsing’ in Firefox most data is sandboxed to the current session and erased after closing the window. LSOs were an exception to this since they are set and read by the Flash plugin and not the browser directly. They persisted on a user’s computer when switching back to a normal browsing session and could be read in private browsing mode even if they were set during the regular session. Flash itself has updated its behavior to respect the privacy mode of various browsers as of version 10.1 [14], but this illustrates how little control browsers have over plugins.

While LSOs themselves are only accessible within a Flash element, the information they contain can be leaked out. The simplest way to accomplish this is through JavaScript, which was first done by the evercookie project [5]. JavaScript on the webpage can communicate with the ActionScript inside the Flash object using just built-in methods. Two functions are defined in ActionScript, one for writing an LSO to JavaScript and one for reading it [Appendix 9.1]. Each function uses the ExternalInterface built into Flash to do so [15]. The JavaScript code can then reference those functions on the embedded Flash object no differently than any other object [Appendix 9.5.4.1]. One important factor in this flow is that the Flash object must be loaded and embedded on the page before it can be referenced by JavaScript.
Since the DOM can be considered to be ready before the page is fully loaded, the user could potentially navigate away to a different page before the LSO objects can be accessed.

### 4.1.3. JavaScript Localstorage

After a client receives the response from its initial HTTP request, the browser executes a block of JavaScript before rendering the page. The Flash object is dynamically embedded in the page by replacing an HTML element. If Flash is blocked, the original content of the element is left intact instead. A timed delay is set to ensure that the Flash object has time to load. Once the delay has completed, the script sends a request to the Flash object for the LSO as discussed above. If there is a stored value already, it will be returned and stored as a JavaScript variable. Once the ActionScript within Flash has returned its result, an HTTP GET request is sent to the web server using AJAX [Appendix 9.5.2.1].

The AJAX query has all of the relevant data set in the URL. It is sent to /ajax/LOCALSTORAGE/LSO. The value of LOCALSTORAGE is pulled from the HTML5 localStorage structure. The value of LSO is taken from the result of the ActionScript function within the Flash object. If either of these values is not available, either because they were blocked or empty, then they are replaced with a “0”. The web server treats this as a placeholder value and will simply ignore it, while anything else is treated as a potential profile ID.

### 4.2. Geolocation

The geolocation of each session is determined by a geographic information system integrated with Django [19]. The IP address of the client is passed in to determine the location of the IP address in latitude and longitude. The spherical distance between this set of latitude and longitude coordinates and that stored in each profile is then calculated [20]. If the two sets of coordinates are within 50 miles of each other, the new session is determined to match the profile being checked [Appendix 9.5.1.4].
Using this method, it is possible that a single IP address can match many profiles based on geolocation. All matches within the specified distance are taken into account when determining what the session matches, not just the closest coordinates. This effectively makes the geolocation values apply to an area of a defined size, which can help account for small movements. Although it usually isn’t possible to get an exact location for IP addresses allocated to a home user, IP addresses are allocated geographically at a larger scale. A user using a dynamic IP address could change IPs on a daily basis but every IP would likely be within the same small block and could be geolocated to the same broad area.

4.3. **Fake DNS**

Clients can leak some information about their network through the DNS server they use. The majority of networks utilize their own DNS servers instead of a generic public server. More importantly, clients rarely change DNS servers. Once a DNS server is set, either statically or by DHCP, the same server is likely to continue resolving names for the client in the future. Matching the address of a DNS resolver with a given profile can be done by forcing the client to make a DNS request that will uniquely identify them.

In the DNS request shown in Figure 2 below, the address of the client’s DNS server was collected by embedding an image in the web page and using a unique implementation of a controlled DNS server [Appendix 9.3]. Using the domain example.com for the web server, the subdomain profiles.example.com was setup with a NS record pointing to the web server. On the web page itself, an image tag was included with a URL referencing the subdomain, http://profileid.profiles.example.com/image.png, where ‘profileid’ represents the actual identifier for the user’s profile.
When a user makes an HTTP request to http://example.com/, the server generates or loads their profile and returns a reference to http://profileid.profile.example.com/image.png in the HTML code for the page [Appendix 9.5.1.3]. The user’s browser automatically performs a DNS lookup for this address and finds the name server for profile.example.com. Since the name server is also the web server, the browser sends a DNS request to the same server that is serving the web content and tracking profiles. Once the web server receives the DNS request, it looks up the value of “profile” in the profile database. If found, it adds the address that the request came from to the profile and returns a DNS response using a modified version of a publicly available minimal DNS server [18]. The response will contain the IP address of the web server regardless of what the actual profile ID in the request is. The browser finally sends an HTTP request to the web server for the image itself. A diagram of this flow can be seen in Figure 3 below.
4.4. **Fingerprinting**

The techniques used for fingerprinting involve determining specific characteristics about the client's environment. Some of these are passively observed from the connection to the server. Other characteristics are actively requested by the server and would otherwise not be sent. Regardless of how these properties are obtained, they provide information about the client without relying on the client to store any data. This information varies in how unique it is and how likely it is to change.

4.4.1. **Browser matching**

If a client makes a request and no profile ID is found in any of the possible storage methods, the server will attempt to match the browser with browsers that have previously made requests. One of the headers normally sent in an HTTP request, although not required, is the User-Agent [9]. This header provides a relatively unique identification string as it often contains full version information for both the browser and operating system, but it can also change very frequently. For instance, something as simple as a minor browser patch can change the information in the User Agent string. However, the string will only be partially changed as a version update to a browser will typically only result in a few characters changing. Partial string matches can be used to determine how closely the old and new User Agents...
match without having to be exact. To make user agent strings a measurable metric even after they have been slightly modified, the last recorded string for a given profile and the newly sent one are compared in all lower case and a ratio of their similarity is calculated [Appendix 9.5.1.4]. This is done using the Ratcliff/Obershelp algorithm which computes the doubled number of matching characters divided by the total number of characters in the two strings [16]. The result is a ratio between 0.0 and 1.0, where 1.0 represents two identical strings. Using this algorithm a minor version update in Chrome, from ‘Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.9 Safari/536.5’ to ‘Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/19.77.34.5 Safari/537.1’, gives a very high ratio of 0.919.

A bigger concern than frequently changing user agents is the ease of which they can be spoofed. Of all browser properties this is the easiest for clients to intentionally change. The main reason being that it is one of the oldest methods for identifying HTTP clients that can be consistently used across all browsers [17]. The privacy implications are well known and many browsers now have built-in tools targeted at developers that can change the user agent to any arbitrary string. Custom browsers and non-interactive HTTP clients are free to set the user agent to anything they want and can even exclude it entirely. As a result, user agents cannot be trusted to be accurate and can only be given a small amount of weight in identifying a specific user. Combined with the small reliability value, only strings that have over a 90% correlation are kept as matches [Appendix 9.5.1.4]. Anything below that is discarded as not being similar enough.

**4.4.2. OS fingerprinting**

Although the operating system that a web client uses is generally not very unique, it is still useful in the overall tracking plan. The OS detected is associated with the user’s profile, but it is given a small enough value that no sessions will be correlated to a profile based just on the OS. Determining
what the OS the client is running is done by using p0f [Appendix 9.2]. p0f is a freely available tool for
performing passive fingerprinting of operating systems [21]. It performs identification by capturing
traffic from the connecting host and comparing certain fields with a database of known signatures. The
factors that the tool looks at include the ordering of TCP options, the maximum segment and windows
sizes, TCP timestamps, TOS flags, and the TTL of packets. This information can be used to identify the
underlying TCP/IP stack with a strong degree of certainty, which in turn identifies the operating system.
Although an OS isn’t a particularly unique piece of data in a profile, it is very reliable. The only way to
prevent the passive fingerprinting from succeeding is to modify the system’s TCP/IP parameters, which
in turn is almost guaranteed to cause an impact on the system’s network performance [22].

In order to hook the web server into the data generated by p0f, a log synchronization script was
developed [Appendix 9.4]. As the p0f log is filled with new entries, a script watching the log parses each
new line. It ignores everything except for TCP packets marked with the SYN flag coming from the client.
Once a packet matching these parameters is found, the detected OS and IP address are read in. They are
then synchronized to the database of profiles, although these values are not matched with any specific
profile since at this point the required HTTP traffic hasn’t yet been generated. Instead, the IP address and
timestamp are entered into a different table created for solely for containing the operating system
information [Appendix 9.5.3.1]. As the client generates HTTP activity, the session will be continuously
checked against this database and correlation is performed based on timestamps and the IP address.
When a match is found the OS that was saved is linked with the current profile.

Although the web server does not receive any HTTP packets until after the TCP three-way
handshake has occurred, in practice the p0f logs typically lag behind the requests being sent to the web
server due to the time needed for p0f to analyze the packets and write to the file system. Even with
extremely bad delays, the time difference will likely not be more than one second since p0f starts
analyzing and writing data before the web server see any requests. If a session matches a detected OS from the past second with the same IP, it is taken as a match [Appendix 9.5.1.4]. The only way for this method to create an inaccurate match is if two or more devices that have the same OS make requests to the server within one second of each other. While this could conceivably occur, there are few methods that would provide a higher level of accuracy. One possibility for improving the accuracy is using the client’s TCP source port. While this is easily accessible from the p0f logs, there is no way to obtain this information within the web framework used. Therefore, time based IP address matching is the best viable option.

4.4.3. System Fonts

The available system fonts can be determined by examining the width of a given string. First, the string is rendered using the default font and the pixel width of the string is measured [Appendix 9.5.2.1]. The same string at the same font size is then rendered with a different specified font, and its width is recorded. If the two lengths differ, then that font is available on the system. The process is repeated for all fonts to be tested [23]. A list of fonts is loaded from a text file on the web server and each is checked in order. Once every font has been tested, an AJAX POST request is sent to the server containing a list of all the fonts that were matched, an example of which can be seen in Figure 4.
The string rendered for each font is “mmmmmmmmmmlli”. The “m” characters provide the greatest width for the string. At the end, the trailing “l” and “i” characters help distinguish mono spaced fonts. Any mono spaced fonts will have the same width for these characters as any other, while the default base font will render them much shorter. The string is first rendered with a default font and its pixel width is measured. Each font from the list of fonts to test is then used in an attempt to render the string, with the default control font listed as a fallback in the HTML code. The width of the each new rendering is measured and compared to the default font. Three font families are tried for each font; monospace, sans-serif and sans. This helps ensure a great amount of accuracy in the detection. If there is a difference in the pixel width, then it can be determined that the font is installed.

System fonts can also be more accurately detected using Flash [24]. Using a standard method in ActionScript, a list of all available fonts on the system can be easily generated [Appendix 9.1]. As with the JavaScript font detection method, once the list of fonts on the system has been created it is sent in a POST request to the tracking server. There are several benefits to using ActionScript detection over JavaScript. The largest benefit is that there is no need to have a list of fonts to check for. The enumeration will find every available font; even custom fonts that would not normally be in an external
list will be detected and recorded. The only factor that limits the reliability of font detection using ActionScript is that Flash must be available on the client’s browser.

Once the list of fonts has been obtained within Flash, a HTTP POST request is generated [Appendix 9.5.2.1]. It is sent with a relative path to the current server, and contains the fonts as POST data. An example of this request, sent from Firefox, can be seen in Figure 5. Flash automatically sends all cookies with HTTP requests as the actual requests are made over the browser; the plugin does not implement its own HTTP stack separately. As a result, the server can store the reported fonts with the user’s current profile without needing any information beyond what the browser sends in the POST request.

![Figure 5](image_url)

**4.4.4. Plugins**

A list of plugins used by the browser, including the specific version and filename of the plugin, is available as a standard property in JavaScript across all major browsers [25]. This list is generated and then sent back to the server through an AJAX POST request to “/plugins” as can be seen in Figure 6 [Appendix 9.5.4.2]. The list is sorted and then matched against existing profiles. With this method, if a plugin is updated to a newer version the entire list will change and no longer match existing profiles.
This makes it easy to declare a non-matching list of plugins as a false negative, but it also makes matching lists more reliable. Since the matching values are aggregated, this is preferable to having less definite but more frequent matches. When there is a match, there is a higher certainty that the match is correct.

There is a high amount of identifiable information available from plugins, primarily from the very detailed version numbers available [26]. Despite this relatively high accuracy of the plugin matches, the prevalence of automatic updating in browsers makes it common for different users to have the same versions across their plugins. Matching values are given a low value of only 4 points in order to offset the potential generic matches that may occur. The plugin matches are most useful in a corporate environment where automatic updates are disabled or for a snapshot of a virtual machine that isn’t fully up to date. Since the plugin’s full version string is easily obtained along with the name of the plugin, there is less likely to be multiple conflicting matches in these situations than when every plugin has been patched to the newest version.

5. Profile algorithm

Depending on what changes between sessions, there is the potential for multiple conflicting profile identifiers to be reported. Determining which of these is the correct one to use is done by assigning a value to each ID based on the source that it came from. More reliable sources have higher
values, while less specific or unreliable sources are given lower values [Appendix 9.5.1.1]. Before the best ID is chosen, any single ID that is reported by multiple sources will have the values summed. The highest value is then picked from the aggregate profiles [Appendix 9.5.1.4].

No profile can be identified with absolute certainty. Any of the places used for identifying information can be spoofed or hidden with various degrees of difficulty. A number is generated from the total aggregated profiles by combining the scores given to each matched tracking method. This number represents the level of certainty that the correct profile has been found and is used for comparison if multiple profiles are matched. There is always a danger of selecting a false negative, but this is reduced as the certainty value increases. Incorrectly identifying a new user as an existing profile is the most frequently occurring cause of false negatives as there is less information available for new users than for repeated sessions. This is unavoidable, but as long as it is controlled and minimized it does not severely degrade the quality of the tracking algorithm.

To help with false negatives, there is a flag set that identifies a profile as being newly created. This allows sessions to be put into an uncertainty state if there is initially only a small amount of information available. Once additional profile information has been collected for that session, one of two things may occur. The first is that the new information matches to a different existing profile; in this case the session is identified as belonging to the matching profile and the temporary profile is deleted. The other possibility is that not enough information is detected in a reasonable amount of time to match the session against an existing profile. If this occurs, the temporary profile has its flag removed and is associated with that session permanently. For testing purposes, this limit was set to three minutes as it only needs to be long enough for the client to load every embedded object and script, run the Flash and JavaScript functions, and send the resulting data back to the server.

When a profile is marked as being temporary or newly created, all identifying data sources have their values reduced by half. This ensures that any new data that comes in matches established profiles.
over the temporary profile. However, if the new information is unique and not yet recorded in the profile database, then cutting its value in half has no effect and it will be associated with the temporary profile. After a period of three minutes, a temporary profile is no longer considered temporary as this is enough time for an active connection to generate nearly all of the identifying information that is being tracked. Once a profile has been created it is permanently stored in a database. Any future sessions that are associated with the profile will update the identifying data for that profile. IP addresses, DNS addresses, and OSes are stored as a list and any old values are maintained. All other parameters are overwritten to the most recent value.

<table>
<thead>
<tr>
<th>Tracking Method</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP cookies</td>
<td>15</td>
</tr>
<tr>
<td>Etags</td>
<td>16</td>
</tr>
<tr>
<td>Flash LSOs</td>
<td>16</td>
</tr>
<tr>
<td>Localstorage</td>
<td>15</td>
</tr>
<tr>
<td>Geolocation</td>
<td>5</td>
</tr>
<tr>
<td>DNS address</td>
<td>2</td>
</tr>
<tr>
<td>User-Agent fingerprinting</td>
<td>2</td>
</tr>
<tr>
<td>OS fingerprinting</td>
<td>1</td>
</tr>
<tr>
<td>System fonts (Flash)</td>
<td>14</td>
</tr>
<tr>
<td>System fonts (JavaScript)</td>
<td>10</td>
</tr>
<tr>
<td>Plugins</td>
<td>4</td>
</tr>
</tbody>
</table>

*Table 1*

The exact weights for all of the analyzed tracking methods can be seen in Table 1 above. The values are assigned based on the expected reliability of a given method. The highest weighted methods are methods that are extremely difficult to spoof; they also provide very unique information to identify
the correct profile. Methods that are less unique or are easier to spoof or change are rated lower. The ability to block a method completely does not affect its weight; only the ability to modify the relevant information will negatively affect the weight.

The reliability of any given method can vary based on the current environment. Using a weighted algorithm allows for adjustments to be easily made to fit the situation. Once it is known which methods are most reliable they can be given new values and the tracking algorithm will behave accordingly. One example of this would be monitoring traffic from a specific company. In this scenario, it is likely that many of the clients will be using the same OS and User-Agents; those factors could be given lower weights, while plugins may be more unique and given a higher weight. The difficulty only lies in determining which methods are most reliable for the given environment.

6. Test results

6.1. Browser behavior

With most modern browsers, there is a distinction made between different ways of reloading a web page that affects how much information is available in the requests. Depending on how the user chooses to refresh the browser it may or may not validate the cache. Validating the browser cache will cause all requests to behave as expected and the traffic will look similar to the initial request. Not validating the cache causes the browser to just send a request for the main resource as well as any AJAX requests embedded in the page. Any static files, such as images or scripts, will not be requested again and as a result no etag will be sent in the request headers.

The combined tracking methods were tested on Chrome 19.0.1084.46, Firefox 13.0, Safari 5.1.7, and Internet Explorer 8 on Windows XP. These were chosen as they are the four most popular browsers and collectively make up a 96.64% of all browsers worldwide [27]. Each browser was tested with an empty database of profiles. After the initial load, the browser’s cookies were cleared and the site was
reloaded. Next the full browser history, including cookies and cache, were cleared and the site was reloaded twice. The purpose of reloading twice instead of just once is to make sure the correct profile is set on the client. Since the server only makes the most intelligent decision after it has all the available data, the initial load may set a temporary profile on the client which persists even after a more appropriate profile is found. Reloading again allows the server to send what it has determined to be the most accurate profile back to the client and replace the temporary values that are stored. The one exception to this set of tests was Safari. Safari does not provide users with a way of distinguishing different kinds of stored browser data. It only allows clearing all data at once for a particular website so it was not possible to test the effect of just clearing the saved cookies.

In each browser tested, the correct profile value was found even after clearing the browser’s cookies and cache. Although there were no values sent to the server from the client, the profile was matched on every passive value. The most identifying matches were the fonts from both JavaScript and Flash, the User-Agent header, and the list of browser plugins. Additionally, the DNS server, geolocation of the client’s IP, and the OS’s TCP/IP properties all matched as well, giving the original profile for every browser a high certainty of matching with a total score of 68. This is well above the selected minimum threshold of 15 needed to identify a matching profile.

To see how much data can be shared between different browsers, a profile was created by loading the tracking site twice in a browser. Each other browser was then used to load the site without resetting the profile database. In each case, there was enough matching information to identify the same profile. Not all of the browsers shared the same properties, however. Since Chrome runs a sandboxed version of Flash it has its own unshared LSO storage. Every other browser did share LSOs between each other and the saved profile value was sent to the server. This is the only locally stored value that was shared among the browsers. HTTP cookies, browser caches, and HTML5 LocalStorage are all unique to each browser and were not accessible from other browsers.
While no container could be used to store values across browsers other than LSOs, some of the identifying browser metadata did carry over. Each browser was able to render the same set of fonts and had matching sets when using both the JavaScript and Flash font detection, indicating that the fonts can be used to identify the client’s system rather than just identifying the browser. Conversely, the list of plugins was unique for each and did not match across different browsers. The User Agent header for the different browsers partially matched each time, but not enough to give a positive correlation. The headers in Figure 7 and Figure 8 show the user agents being sent by Chrome and Safari.

**Figure 7**

GET http://tracker.loosepackets.com/ HTTP/1.1
Host: tracker.loosepackets.com
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.46 Safari/536.5
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1;utf-8;q=0.7,*;q=0.3

**Figure 8**

GET http://tracker.loosepackets.com/ HTTP/1.1
Host: tracker.loosepackets.com
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/534.57.2 (KHTML, like Gecko) Version/5.1.7 Safari/534.57.2
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: keep-alive

These two browsers use the same rendering engine and as a result have similar user agent strings. Despite the close similarities, there is only an 85% correlation using the Ratcliff/Obershulp algorithm which is under the 90% ratio needed to confirm a pair of matching user agents. Firefox and Internet Explorer have user agents that are very unique and have a much lower correlation to that of other browsers.

### 6.2. Summary of methods

Table 2 below shows a summary of the effectiveness and resilience of each tracking method.
### Table 2

<table>
<thead>
<tr>
<th>Tracking Method</th>
<th>Cross-browser</th>
<th>Uniqueness</th>
<th>Frequency of change</th>
<th>Persists after clearing cache/cookies</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP cookies</td>
<td>No</td>
<td>Very high</td>
<td>Medium</td>
<td>No</td>
</tr>
<tr>
<td>Etags</td>
<td>No</td>
<td>Very high</td>
<td>Low</td>
<td>No</td>
</tr>
<tr>
<td>Flash LSOs</td>
<td>Yes</td>
<td>Very high</td>
<td>Low</td>
<td>Yes but the behavior will change in future versions</td>
</tr>
<tr>
<td>Localstorage</td>
<td>No</td>
<td>Very high</td>
<td>Medium</td>
<td>No</td>
</tr>
<tr>
<td>Geolocation</td>
<td>Yes</td>
<td>Medium</td>
<td>Medium</td>
<td>Yes</td>
</tr>
<tr>
<td>DNS address</td>
<td>Yes</td>
<td>Low</td>
<td>Medium</td>
<td>Yes</td>
</tr>
<tr>
<td>User-Agent fingerprinting</td>
<td>No</td>
<td>Medium</td>
<td>Medium</td>
<td>Yes</td>
</tr>
<tr>
<td>OS fingerprinting</td>
<td>Yes</td>
<td>Low</td>
<td>Low</td>
<td>Yes</td>
</tr>
<tr>
<td>System fonts</td>
<td>Yes</td>
<td>Medium</td>
<td>Low</td>
<td>Yes</td>
</tr>
<tr>
<td>Plugins</td>
<td>No</td>
<td>Medium</td>
<td>Low</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 6.3. Existing vendor solution

For comparison purposes, a tracking appliance of a leading vendor was tested to determine what methods it employed. The MyKonos appliance provides two related services to web servers. It attempts to detect attacks against the web server, and tracks known attackers for future actions. The two services are directly linked; tracking only begins after an attack has been detected.

The MyKonos appliance does not use IP addresses for any tracking; it simply logs them as environmental information. Changing a client’s IP address does not cause the attacker to disappear. A new session is created, but that session is correlated to the attacker’s profile. The actual tracking of the attacker is done through various client-side storage methods, many of which were discussed previously. A tracking cookie is created that indicates the profile of the current session. Similar to changing IP addresses, deleting the tracking cookie does not generate a new profile. The same information is stored in Flash LSOs and Silverlight Isolated Storage. Although these storage objects don't have a built-in expiration mechanism, MyKonos puts a configurable expiration date into the value; even though the
storage object may never be deleted, the value can be discarded as being too old after it has been evaluated by the appliance.

Environmental data, such as the User-Agent string or IP address, is gathered and stored with a given profile; however this data is not used for any actual request correlation or tracking. Since the appliance relies solely on client storage, it is not possible to track any automated tools or correlate them to an existing browser-based session. It also makes it trivial for a client to bypass the tracking, particularly with the increased control over Flash storage that is planned for the next few browser versions. Any match that is detected, however, is very likely to be accurate due to the difficulty of faking the session data.

6.4. **Defeating the tracking methods**

The strength of the profiling algorithm comes from its ability to use many tracking methods in conjunction; it does not rely on any one method in order to accurately track end clients. As a result, defeating the profile detection requires advanced knowledge of the methods being used as each one must be bypassed individually. Blocking any single method will have little effect on the overall effectiveness of the tracking. Assuming that an end client were aware of the exact tracking method used, it is difficult but possible to work around all of the methods.

There are several options available as configurable settings within most browsers to limit the effectiveness of these tracking methods. Disabling JavaScript in the browser is the most effective way to limit the tracking abilities as it affects several methods at once. The ability to detect fonts with JavaScript, storing values in the DOM local storage, and detecting installed browser plugins all require that the browser has JavaScript enabled. Standard HTTP cookies can also be blocked by the browser, causing the cookie to never get stored or sent back to the server. Both of these approaches will severely limit the functionality of most web sites, but are completely effective at preventing these particular web
tracking methods. Fingerprinting based on the User-Agent can be prevented by changing or removing the User-Agent from requests. Since this HTTP header is not required, there is little harm in a client removing it for all websites as a precaution. Etag tracking relies on the same limitation; it is a HTTP header that is commonly stored and sent by web clients, but they are not required to. Disabling caching on browsers will prevent this particular storage mechanism from being utilized.

Flash LSOs and Flash font detection each require Adobe Flash to be installed and enabled on the browser. Flash does not provide fine-grained permission controls or settings, so disabling Flash completely is the only way to defeat these methods. Chrome in particular requires some additional configuration as it has its own built-in version of Flash; removing Flash from the system will still allow these methods to work in Chrome. Flash would need to be disabled in Chrome separately as well.

Geolocation, OS fingerprinting, and DNS resolver detection are all methods that cannot be completely blocked. They exploit fundamental properties of a web transaction in order to determine unique information about a client. However, they can be spoofed to present the server with incorrect values. Geolocation is based on IP addresses so connecting from behind a proxy server will cause the server to perform a geolocation lookup on the wrong IP. The client would need to frequently change to a proxy in a different geographic area in order for this to be effective, however. OS fingerprinting looks at the basic TCP/IP properties in the HTTP connection. On most operating systems, it is possible to change these properties to match that of a different OS. Similar to using a proxy, a client would need to frequently change these properties in order to prevent having a profile built against it. The DNS detection also relies on the basic functionality of web traffic, and a client could only spoof this information by frequently changing name servers. Each time it did, the server would still be able to detect the client's new DNS resolver; it would not be able to correlate the requests to previous ones, however.
7. Conclusions

None of the techniques discussed here for tracking and profiling users are new or unknown. In fact, some vendors have already begun developing preventive measures for the more heavily publicized ones, particularly Flash LSOs. As of Flash version 10.3, browsers are able to delete Flash LSOs directly [28]. Firefox implements this feature as of version 4 and will delete LSOs when it clearing HTTP cookies [29]. Microsoft implemented this same functionality in Internet Explorer 8 [30], and Chrome implemented it in version 12 [31]. Firefox developers have also provided the ability to require user interaction to load any plugin, similar to how Android mobile phones load Flash, and may have this enabled by default in future versions [32].

There are other techniques that can be used to identify connecting HTTP hosts which were not implemented here. One of the possibilities that is difficult to implement and unreliable but potentially very accurate is Java applet profiling. Java applets can provide a great deal of information about the client since they can interact directly with the hardware. However, unlike many of the methods implemented here, Java applets are extremely difficult to hide. Java requires the user to explicitly run the plugin when it loads in Chrome [33], and Internet Explorer can be configured for similar behavior across all websites or with more fine-grained detail [34]. Java support across browsers has also been trending downwards [35], making it less likely to appear on websites and more suspicious when a user encounters a site requiring a Java applet to run.

The profiling algorithm could also be improved by the use of a statistical model to determine the weights of the tracking methods. Deriving such a model was outside the scope of this work, but would likely improve the accuracy greatly. In order to be effective, the statistical model would need to adjust to the environment being monitored; this would allow the weights of each method to fluctuate without the need to manually set them.
The process used here highlights the difficulty in creating true anonymity despite the preventive measures being implemented by browsers. Even with many of the methods blocked or otherwise rendered useless, there is enough certainty in the available information to positively and accurately identify a given system. While some of the methods used are relatively easy to block, such as disabling Flash, many of them rely on passively transmitted information that is impossible to block and extremely difficult to intentionally modify. As new ways of leaking identifying information are discovered, they can be plugged into an algorithm similar to the one discussed here with ease. Browsers are forced to be reactive, developing ways to prevent tracking methods well after they are publicly known. Additionally, older methods can still be used on hosts that support them with the same reliability as hard to implement cutting-edge methods. This multi-pronged approach allows for adaptability to any foreseeable environment.
8. Bibliography


9. Appendix

I authored all of the following code using a combination of Python and Adobe ActionScript. The ActionScript code was compiled and placed in the static directory of the web application as described in section 9.5.4. The Python code was started with the *start.py* script from section 9.2 below.

9.1. **ActionScript code**

This code is embedded in a Flash object that has no visible objects.

```plaintext
import flash.external.ExternalInterface;
import flash.net.URLRequest;
import flash.net.URLVariables;
import flash.text.Font;

//call from javascript
ExternalInterface.addCallback( "requestLSOValue", requestLSOValue );
ExternalInterface.addCallback( "setLSOValue", setLSOValue );

function requestLSOValue():void
{
    var obj_value:String = '0';
    var dataStore = SharedObject.getLocal("profile");

    // If name variable is not available, we assume that this is a new user
    if (dataStore.data.value!=null)
    {
        obj_value = dataStore.data.value;
    }
```

```
function setLSOValue( obj_value ):void
{
    var writeSuccess:Boolean = false;
    // Create a shared-object named "userData"
    var dataStore = SharedObject.getLocal("profile");
    // Store the name
    dataStore.data.value=obj_value;
    var success = dataStore.flush();
    if (success == "pending") {
        dataStore.onStatus = function(result) {
            if (result.code == "SharedObject.Flush.Success") {
                writeSuccess = true;
            } else {
                writeSuccess = false;
            }
        };
    } else {
        writeSuccess = success;
    }
}

function GetAllTheFonts()
{
var all_the_fonts = Font.enumerateFonts(true);

// Create a HTTP request
var loader : URLLoader = new URLLoader();
var request : URLRequest = new URLRequest("/flash/fonts/");
request.method = URLRequestMethod.POST;
var variables : URLVariables = new URLVariables();

var font_names = new Array();
for (var i:Number=0; i<all_the_fonts.length; i++)
{
    font_names.push(all_the_fonts[i].fontName);
}

variables.flash_fonts = font_names;

request.data = variables;
loader.load(request);

stop();
}

GetAllTheFonts();

9.2. start.py
This script is used to start all components at once, including the p0f log parser, DNS responder, and
Django web server.

```python
from subprocess import Popen
import sys
import signal

port = '80'
domain = "profile.tracker.loosepackets.com."
ip = "23.23.105.113"
logfile = '/var/log/p0f.log'

def signalhandler(signal, frame):
    dns_proc.kill()
    print
    print "DNS responder stopped"
Popen("pkill p0f", shell=True)
    print "p0f stopped"
p0f_parse_proc.kill()
    print "p0f parser stopped"
django_proc.kill()
    print "Django stopped"
sys.exit()

signal.signal(signal.SIGINT, signalhandler)
socket = '0.0.0.0:%s' % port

dns_args = [sys.executable, 'dns_track.py', domain, ip]
```
django_args = [sys.executable, 'webtracking/manage.py', 'runserver', socket]

p0f_args = "'/usr/sbin/p0f -f /etc/p0f/p0f.fp -d -i eth0 -o %s 'tcp and port 80'" % logfile

p0f_parse_args = [sys.executable, 'p0f_parse.py', logfile]

dns_proc = Popen(dns_args)
print "DNS responder started with PID of %s" % dns_proc.pid

p0f_proc = Popen(p0f_args, shell=True)
print "p0f started with PID of %s, logging to %s" % (p0f_proc.pid, logfile)

p0f_parse_proc = Popen(p0f_parse_args)
print "p0f log parser started with PID of %s" % p0f_parse_proc.pid

django_proc = Popen(django_args)
print "Django started on port %s with PID of %s" % (port, django_proc.pid)

print

while True:
    continue

9.3. **dns_track.py**

This script is used to respond to incoming DNS requests.
# Partial code taken from http://code.activestate.com/recipes/491264-mini-
fake-dns-server/
from django.core.management import setup_environ
import sys
from webtracking import settings
setup_environ(settings)

from webtracking.profiles.models import Profile, IPAddress
from twisted.internet.protocol import DatagramProtocol

class DNSQuery(DatagramProtocol):
    def __init__(self, domain, ip):
        self.topdomain = domain
        # Convert the IP address to hex
        self.ip = str.join('', map(lambda x: chr(int(x)), ip.split('.')))
        print "Responding to tracking requests for domain '%s'" % 
        self.topdomain

    def datagramReceived(self, data, (host, port)):
        self.data = data
        self.domain = ""
        self.remote = (host, port)

        opcode = (ord(data[2]) >> 3) & 15   # Opcode bits
        if opcode == 0:
            ini=12
            lon=ord(data[ini])
while lon != 0:
    self.domain += data[ini+1:ini+lon+1]+'.'
    ini+=lon+1
    lon=ord(data[ini])

request = self.domain.split(".")
if ".".join(request[1:]) == self.topdomain:
    self.posrespond()

print "Request: %s from %s" % (request[0], self.remote[0])

try:
    profile = Profile.objects.get(name=request[0])
    print "Found profile for %s" % request[0]

    ipaddr, newip = IPAddress.objects.get_or_create(addr=self.remote[0])
    if newip:
        ipaddr.save()

        profile.dnsaddrs.add(ipaddr)

except:
    print "No profile matches %s" % request[0]
    #self.negrespond()
else:
    print "Wrong domain"
    self.negrespond()
def posrespond(self):
    packet=''
    if self.domain:
        packet += self.data[:2] + '\x81\x80'
        # Questions and Answers Counts
        packet += self.data[12:]
        # Original Domain Name Question
        packet += '\xc0\xc0'
        # Pointer to domain name
        packet += '\x00\x01\x00\x01\x00\x00\x00\x3c\x00\x04'  # Response type, ttl and resource data len -> 4 bytes
        packet += self.ip
        # 4bytes of IP
        self.transport.write(packet, self.remote)

def negrespond(self):
    packet = ''
    self.transport.write(packet, self.remote)

if __name__ == '__main__':
    from twisted.internet import reactor

    domain = sys.argv[1]
    ip = sys.argv[2]
reactor.listenUDP(53, DNSQuery(domain, ip))
reactor.run()

9.4. **p0f_parse.py**

This file monitors and parses the p0f log, updating the profile database when appropriate data is found.

```python
# Monitors the p0f log for new lines, and parses them into the Django database
#
# Elements in the attribute dictionary that we care about:
#   cli - IP address and port of the client
#   os  - operating system fingerprint
#   dist - number of hops to reach the device
#   mod - subsystem that generated the entry
#   app - browser fingerprint
#   subj - whether the client or server are being fingerprinted

import time
import datetime
import os
import sys

# Needs to be before any Django imports
os.environ['DJANGO_SETTINGS_MODULE'] = "webtracking.settings"

from webtracking.profiles.models import OperatingSystem, IPAddress, Profile
```
def watch(name):
    
    file = open(name, 'r')
    
    # If there is a new line in the file, that line gets yielded.
    # If not then the function sleeps for 0.5 seconds.
    # This will start by reading all existing lines in the file.
    while True:
        
        new = file.readline()
        
        if new:
            yield new
        else:
            time.sleep(0.5)


def parseLine(line):
    
    split_line = line.split('|')
    
    split_line[-1] = split_line[-1].rstrip()
    
    timestamp = split_line[0].split(' ')
    try:
        datestr = timestamp[0].lstrip('[')
        timestr = timestamp[1].rstrip(']')
        split_line[0] = timestamp[2]
    except IndexError:
        
        print '\nBad timestamp: %s' % str(timestamp)
        return

    attributes = {}
for item in split_line:
    item = item.split(‘=’)
    attributes[item[0]] = item[1]

# The client fingerprinting is the only part that matters.
# However, filtering the captured packets to only the client section
# prevents browser identification.
if attributes[‘subj’] == ‘cli’:
    if attributes[‘mod’] == ‘syn’:
        ip, port = attributes[‘cli’].split(‘/’)

        # Create or get the OS object, then add this IP to it
        ipaddr, new_ip = IPAddress.objects.get_or_create(addr=ip)
        if new_ip:
            ipaddr.save()

        os, new_os = OperatingSystem.objects.get_or_create(name=attributes[‘os’], ip_addr=ipaddr)
        os.save()

if __name__ == “__main__”: p0f_log = sys.argv[1]

    for line in watch(p0f_log):
        parseline(line)
9.5. **Django code**

9.5.1. Root files

9.5.1.1. settings.py

```python
# Django settings for webtracking project.
import os.path

DEBUG = True
TEMPLATE_DEBUG = DEBUG

DNSURL = '.profile.tracker.loosepackets.com/static/small.png'

ADMINS = ( )

TRACK = {
    'cookie': 15,
    'dns': 2,
    'flash': 16,
    'etag': 16,
    'localstorage': 15,
    'js_fonts': 10,
    'flash_fonts': 14,
    'os': 1,
    'plugins': 4,
    'geoloc': 5,  # Multiplied inversely by distance
    'user_agent': 2,
}
```
TEMP_TTL = 3
SCORE_THRESHOLD = 15
MANAGERS = ADMINS
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3', # Add 'postgresql_psycopg2',
        'postgresql', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': os.path.join(os.path.dirname(__file__),
            'webtracking.db').replace('\','/'), # Or path to database file if
        using sqlite3.
        'USER': '', # Not used with sqlite3.
        'PASSWORD': '', # Not used with sqlite3.
        'HOST': '', # Set to empty string for localhost.
        Not used with sqlite3.
        'PORT': '', # Set to empty string for default.
        Not used with sqlite3.
    }
}

# Local time zone for this installation.
TIME_ZONE = 'America/Los_Angeles'

# Language code for this installation.
LANGUAGE_CODE = 'en-us'
SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = True

# If you set this to False, Django will not format dates, numbers and
# calendars according to the current locale
USE_L10N = True

# Absolute file system path to the directory that will hold user-uploaded
# files.
MEDIA_ROOT = ''

# URL that handles the media served from MEDIA_ROOT.
MEDIA_URL = ''

# Absolute path to the directory static files should be collected to.
STATIC_ROOT = ''

# URL prefix for static files.
STATIC_URL = '/static/'

# URL prefix for admin static files -- CSS, JavaScript and images.
ADMIN_MEDIA_PREFIX = '/static/admin/

# Additional locations of static files
STATICFILES_DIRS = (/
    os.path.join(os.path.dirname(__file__), "static").replace('\\', '/),
# List of finder classes that know how to find static files in various locations.
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
)

SECRET_KEY = 'xva-02@esdt)3s$5w2nzj8rv%y8q%z@o)8j8y=u%y7ho@+xkn'

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
)

MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
)

ROOT_URLCONF = 'webtracking.urls'

TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), "templates").replace('\','/'),
)

TEMPLATE_CONTEXT_PROCESSORS = (}
"django.contrib.auth.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media",
"django.core.context_processors.static",
"django.contrib.messages.context_processors.messages",
)

INSTALLED_APPS = (  
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'webtracking.profiles',
)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format': '%(asctime)s: %(message)s'
        }
    },
    'handlers': {
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'formatter': 'standard',
        },
        'profile_data': {

        },
    },
}
'level': 'INFO',
'class': 'logging.FileHandler',
'filename': os.path.join(os.path.dirname(__file__),
        'logs/profile_decisions.log').replace('\\','/'),
'formatter': 'standard',
}
},
'loggers': {
    'django.request': {
        'handlers': ['mail_admins'],
        'level': 'ERROR',
        'propagate': True,
    },
    'tracking.profiles': {
        'handlers': ['profile_data'],
        'level': 'INFO',
        'propagate': False,
    }
}
}
}
}

GEOIP_PATH = '/usr/lib/geoip'

9.5.1.2.  urls.py

from django.conf.urls.defaults import patterns, include, url
from django.contrib.auth.views import login, logout
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

from webtracking.views import *

urlpatterns = patterns('',
    (r'^$', mainpage),
    (r'^meta/$', displaymeta),
    (r'^cookies/$', displaycookies),
    (r'^reset/$', resetprofile),
    (r'^showall/$', showall),
    (r'^small.png$', etagcheck),
    (r'^resetallthethings/$', resetdatabase),
    (r'^ajax/(.*)/(.*)', ajaxrequest),
    (r'^fonts/', fontlist),
    (r'^plugins/', plugins),
    (r'^flash/fonts/', flashfonts),
)

urlpatterns += staticfiles_urlpatterns()


9.5.1.3. views.py

from django.http import Http404, HttpResponse
from django.shortcuts import render_to_response, redirect
from django.views.generic.simple import redirect_to
from django.views.decorators.csrf import csrf_exempt
from django.conf import settings
from profiler import profile_check, profile_reset, getallprofiles, deleteprofile, deleteallprofiles

# test functions

@profile_check
def displaymeta(request):
    values = request.META.items()
    values.sort()
    return render_to_response('meta.html',
    {'profile':request.session['cookie'], 'headers':values})

@profile_check
def displaycookies(request):
    values = request.COOKIES.items()
    values.sort()
    return render_to_response('meta.html',
    {'profile':request.session['cookie'], 'headers':values})

def etagger(decorated):
    def wrapper(request):
        if request.META.get('HTTP_ETAG', False):
            etag = request.META['HTTP_ETAG']
        elif request.META.get('HTTP_IF_NONE_MATCH', False):

etag = request.META['HTTP_IF_NONE_MATCH']
else:
    etag = ""

if etag:
    request.session['etag'] = etag

return decorated(request)

return wrapper

@etagger
@profile_check
def mainpage(request):
    return render_to_response('base.html',
    ({'profile':request.session['cookie']}))

@profile_check
@profile_reset
def resetprofile(request):
    return redirect_to(request, url='/', permanent=False)

@profile_check
def showall(request):
    profileset = getallprofiles()
    return render_to_response('allprofiles.html',
    ({'profile':request.session['cookie'], 'profileset':profileset}))
@etagger
@profile_check
def etagcheck(request):
    if 'dns' in settings.TRACK:
        # Redirect to subdomain, which will generate a DNS request
        url = 'http://' + str(request.session['cookie']) + settings.DNSURL
    else:
        url = '/static/small.png'
    response = redirect(url)
    response['Etag'] = request.session['cookie']
    response['Must-Revalidate'] = 1
    return response

@profile_check
def ajaxrequest(request):
    return HttpResponse(request.session['cookie'])

@csrf_exempt
@profile_check
def fontlist(request):
    return HttpResponse(request.session['cookie'])

@csrf_exempt
@profile_check
def plugins(request):
    return HttpResponse(request.session['cookie'])

@csrf_exempt
@profile_check
def flashfonts(request):
    return HttpResponse(request.session['cookie'])

def resetdatabase(request):
    deleteallprofiles()
    request.session.clear()
    return render_to_response('base.html', {'profile':"Unknown"})

9.5.1.4. profiler.py

import datetime
from difflib import SequenceMatcher
import logging
import math
from operator import itemgetter
from os import path
import random
import string
from django.contrib.gis.utils import GeoIP
from django.core.exceptions import ObjectDoesNotExist

from settings import TRACK, TEMP_TTL, SCORE_THRESHOLD
from profiles.models import Profile, IPAddress, Browser, OperatingSystem, Plugin, UserAgent

logger = logging.getLogger('tracking.profiles')

def NewProfile():
    randstr = ''.join(random.choice(string.lowercase) for i in xrange(5))
    name = randstr
    try:
        Profile.objects.get(name=name)
    except Profile.DoesNotExist:
        profile = Profile.objects.create(name=name)
        profile.temp = True
        profile.save()
    return profile

return NewProfile()

def aggregate_profiles(profiles, temp):
    profile_metrics = {}
    # Special handling for geolocation, everything else handled identically
    for method in profiles:
        # Make everything a list to iterate through

if method != 'geoloc':
    if type(profiles[method]) != list:
        profiles[method] = [profiles[method]]

    for name in profiles[method]:
        current_val = profile_metrics.get(name, 0)
        profile_metrics[name] = current_val + TRACK[method]

else:
    for name in profiles[method].keys():
        current_val = profile_metrics.get(name, 0)
        profile_metrics[name] = current_val + (TRACK[method] / profiles[method][name])

if temp and 'cookie' in profiles.keys():
    name = profiles['cookie'][0]
    profile_metrics[name] = profile_metrics[name] / 10.0
    logger.info('%s is a temporary profile' % name)

# Check for an empty object. if it is empty, create a new profile.
if not profile_metrics:
    logger.info("Nothing found, creating a new profile")
    profile = NewProfile()
    return profile
else:
    logger.info("%s
%s" % (str(profiles), str(profile_metrics)))

# This isn't definitive. Multiple profiles could have the same value.
best_match = max(profile_metrics.iteritems(), key=itemgetter(1))[0]
Make sure that the profile actually exists. If it doesn't, burn the profile out of the metrics list with fire.

logger.info("Trying to find %s (%s) in the database..." % (best_match, profile_metrics[best_match]))

try:
    profile = Profile.objects.get(name=best_match)
    logger.info("Profile found."")
    if not profile.temp and profile_metrics[best_match] <
    SCORE_THRESHOLD:
        return NewProfile()
    else:
        return profile
except ObjectDoesNotExist:
    for method in profiles.keys():
        if method == 'geoloc' and best_match in
        profiles[method].keys():
            del(profiles[method][best_match])
        elif best_match in profiles[method]:
            spot = profiles[method].index(best_match)
            del(profiles[method][spot])

    logger.info("Doesn't exist.")

    return aggregate_profiles(profiles, temp)

def getallprofiles():
    allprofiles = {} 
    for profile in Profile.objects.all():
addrs = "<br>).join([str(ip.addr) for ip in profile.ipaddrs.all()])

agents = "<br><br>).join([str(browser.useragent) for browser in profile.browsers.all()])

allprofiles[profile.name] = [addrs, agents]

return allprofiles

def deleteprofile(profilename):
    try:
        temp = Profile.objects.get(name=profilename)
        tempips = temp.ipaddrs.all()
        temp.delete()
    except Profile.DoesNotExist:
        tempips = []

    #Loop through the IPs of the deleted profile
    for ip in tempips:
        if not Profile.objects.filter(ipaddrs=ip):
            #This IP is no longer associated with a profile, delete it
            ip.delete()

def deleteallprofiles():
    allprofiles = Profile.objects.all()
    for i in allprofiles:
        i.delete()
allbrowsers = Browser.objects.all()
for i in allbrowsers:
    i.delete()

def UpdateMetadata(profile, meta):
    return
    
    # Add IP address to the profile.
    addr, newaddr = IPAddress.objects.get_or_create(addr=meta['REMOTE_ADDR'])
    if newaddr:
        addr.save()
    profile.last_client_ip = addr
    profile.ipaddrs.add(addr)

    # Add browser metadata to the profile.
    browser, newbrowser = Browser.objects.get_or_create(
        charset=meta['HTTP_ACCEPT_CHARSET'],
        encoding=meta['HTTP_ACCEPT_ENCODING'],
        language=meta['HTTP_ACCEPT_LANGUAGE'])
    if newbrowser:
        browser.save()
        profile.browsers.add(browser)

    # Set the User-Agent
    useragent, newagent = UserAgent.objects.get_or_create(
        name=meta['HTTP_USER_AGENT'])
    if newagent:
        useragent.put()
profile.useragent.add(useragent)

profile.save()

# Modified from http://www.johndcook.com/python_longitude_latitude.html

def CalculateDistance(lat1, long1, lat_long):
    # Convert latitude and longitude to
    # spherical coordinates in radians.
    degrees_to_radians = math.pi / 180.0

    # phi = 90 - latitude
    ph1 = (90.0 - lat1) * degrees_to_radians
    ph2 = (90.0 - lat_long[0]) * degrees_to_radians

    # theta = longitude
    the1 = long1 * degrees_to_radians
    the2 = lat_long[1] * degrees_to_radians

    # Compute spherical distance from spherical coordinates.

    # For two locations in spherical coordinates
    # (l, theta, phi) and (l, theta, phi)
    # cosine( arc length ) =
    # sin phi sin phi' cos(theta-theta') + cos phi cos phi'
    # distance = rho * arc length

    cos = (math.sin(ph1) * math.sin(ph2) * math.cos(the1 - the2) +
           math.cos(ph1) * math.cos(ph2))
arc = math.acos(cos)

# Remember to multiply arc by the radius of the earth
# in your favorite set of units to get length.
return arc * 38960  # Returns distance in miles

def profile_check.decorated):
    def wrapper(request, local="0", lso="0"):
        logger.info("Checking profiles...")

        # Check if the incoming url contained valid localStorage
        # and LSO values
        if lso != '0':
            request.session['flash'] = lso
        if local != '0':
            request.session['localstorage'] = local

        # Font detection, both from Flash and JS
        js_fonts = ','.join(sorted(request.POST.getlist('js_fonts[]')))
        if js_fonts:
            font_matches = Profile.objects.filter(js_fonts=js_fonts)
            request.session['js_fonts'] = []
            for match in font_matches:
                request.session['js_fonts'].append(match.name)
flash_fonts =
'',.join(sorted(request.POST.getlist('flash Fonts'))))
if flash_fonts:
    font_matches =
Profile.objects.filter(flash_fonts=flash_fonts)
    request.session['flash_fonts'] = []
    for match in font_matches:
        request.session['flash_fonts'].append(match.name)

# JS plugin detection
plugins = '',.join(sorted(request.POST.getlist('js_plugins[]')))  
if plugins:
    plugin_matches = Profile.objects.filter(plugins=plugins)
    request.session['plugins'] = []
    for match in plugin_matches:
        request.session['plugins'].append(match.name)

# Check for OS
ip = request.META.get('REMOTE_ADDR')
request.session['geoloc'] = {}  
if ip:
    ip_addr, new_ip = IPAddress.objects.get_or_create(addr=ip)
    if new_ip:
        ip_addr.save()
    os =
    OperatingSystem.objects.filter(ip_addr=ip_addr).order_by('-last_activity')[0]
    profiles = Profile.objects.filter(last_client_os = os)
request.session['os'] = [profile.name for profile in profiles]

# Geolocate IP address
geo = GeoIP()

location = geo.lat_lon(ip)
if location:
    for temp_profile in Profile.objects.all():
        distance = CalculateDistance(temp_profile.latitude, temp_profile.longitude, location)
        if distance > 50:
            request.session['geoloc'][temp_profile.name] = distance

# Compare User-Agent
user_agent = request.META.get('HTTP_USER_AGENT', None)
if user_agent:
    request.session['user_agent'] = []
    user_agent = user_agent.lower()
    for temp_profile in Profile.objects.all():
        for stored_user_agent in temp_profile.useragent.all():
            ratio = SequenceMatcher(None, stored_user_agent, user_agent).ratio()
            if ratio > 0.9:  # Keep anything with over 90% match as a potential profile
                request.session['user_agent'].append(temp_profile.name)
break

profilesfound = {}
for type in TRACK:
    if request.session.get(type, False):
        profilesfound[type] = request.session[type]

# find best profile match out of all available methods
try:
    old_profile = Profile.objects.get(name=request.session['cookie'])
    cutoff = datetime.datetime.now() -
    datetime.timedelta(minutes=TEMP_TTL)
    if cutoff > old_profile.created:
        old_profile.temp = False
        old_profile.save()
    temp = old_profile.temp
except (ObjectDoesNotExist, KeyError):
    temp = True

profile = aggregate_profiles(profilesfound, temp)

# Delete previous profile if it was temporary
try:
    if old_profile.temp and old_profile != profile:
        old_profile.delete()
except NameError:
    pass
UpdateMetadata(profile, request.META)

# Check if the passed in font list matches the font list for this profile.
# If it doesn't or if there is no current font list, add the fonts in.
if js_fonts and (not profile.js_fonts or profile.js_fonts != js_fonts):
    profile.js_fonts = js_fonts

if flash_fonts and (not profile.flash_fonts or profile.flash_fonts != flash_fonts):
    profile.flash_fonts = flash_fonts

# Add the OS to the profile
if ip:
    profile.last_client_os = os
    profile.os.add(os)

# Add the plugins to the profile
if plugins:
    profile.plugins = plugins

if location:
    profile.latitude = location[0]
    profile.longitude = location[1]

profile.save()
request.session['cookie'] = profile.name
logger.info('Finished checking profile\n')
return decorated(request)

return wrapper

def profile_reset(decorated):
    def wrapper(request):
        #delete the stored profile
        deleteprofile(request.session['cookie'])

        #generate new profile
        profile = NewProfile()
        UpdateMetadata(profile, request.META)
        request.session['cookie'] = profile.name
        request.session['etag'] = None
        return decorated(request)
    return wrapper

9.5.2. /templates directory

9.5.2.1. base.html

{% load static %}
{% load cache %}
{% get_static_prefix as STATIC_PREFIX %}
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=iso-8859-1" />
    <title>{{ block title %}}Profile Tracker{% endblock %}</title>
</head>

{% block scripts %}
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
<script src="{{ STATIC_PREFIX }}swfobject.js"></script>
<script src="{{ STATIC_PREFIX }}flashlso.js"></script>
<script src="{{ STATIC_PREFIX }}plugindetect.js"></script>
<script src="{{ STATIC_PREFIX }}fontdetect.js"></script>
<script type="text/javascript">
    function Initialize() {
        // flash things
        swfobject.embedSWF("{{ STATIC_PREFIX }}flashlogo.swf", flash_vars.moviename, "45", "18", "11");
        var timer = setTimeout("requestLSOValue();", 1500);

        // send localstorage and lso values over an ajax request
        var timer2 = setTimeout("SendAJAX();", 1700);

        $.get("{{ STATIC_PREFIX }}font_list.txt", function(data) {
            // Font detection
            var font_detective = new Detector();
        });

```
var exists = new Boolean();
var fonts_found = new Array();

var font_list = data.split("\n");
if (font_list.length == 1) font_list = data.split("\r\n");

for (index in font_list) {
    font = font_list[index];
    exists = font_detective.detect(font);
    if (exists) fonts_found.push(font);
}
$.post("fonts/", {'js_fonts':fonts_found});

// Detect installed plugins
var plugins_found = DetectPlugins();
$.post("plugins/", {'js_plugins[]':plugins_found});

function SendAJAX() {
    var lso = "0";
    var local = "0";

    if(flash_vars.profileval) lso = flash_vars.profileval;
    if(localStorage.profile) local = localStorage.profile;

    console.log("LSO is %s" % lso);
    console.log("localStorage is %s" % local);
var url = "ajax/" + local + "/" + lso;
$.get(url, function(data) {
    localStorage.profile = data;
    setLSOValue(data);
    $('#profiletext').text(data);
});
</script>
{% endblock %}

<link rel="stylesheet" href="{{ STATIC_PREFIX }}style.css" type="text/css" media="screen" />
</head>

<body onLoad="Initialize();">
<div id="thewholepage">
    <div id="lefthead">
        &nbsp;&nbsp;<div id="profiletext">{{ profile }}</div>
        &nbsp;&nbsp;<a href="showall">View All</a>
        <br>
        &nbsp;&nbsp;<a href="reset">Reset Profile</a>
        <a href="resetallthethings">Reset Everything</a>
    </div>
    <br><br><hr><br>
    <div id="flashlogo"></div>
</div>
</div>
<br><br><hr><br>
<br><br><hr><br>
<br><br><hr><br>
<br><br><div id="flashlogo"></div>
9.5.2.2.  meta.html

{% extends "base.html" %}

{% block title %}Meta Info{% endblock title %}

{% block maincontent %}
<table>
{% for key,value in headers %}
<tr><td>{{ key }}</td><td>{{ value }}</td></tr>
{% endfor %}
</table>
{% endblock maincontent %}

9.5.2.3.  allprofiles.html
{% extends "base.html" %}

{% block title %}All Profiles{% endblock title %}

{% block maincontent %}
<table border="1">
<tr><th>Profile Name</th><th>IP Addresses</th><th>User Agent</th></tr>
{% for name,values in profileset.items %}
<tr><td>{{ name }}</td>
    {% for item in values %}
        <td>{{ item }}</td>
    {% endfor %}
</tr>
{% endfor %}
</table>
{% endblock maincontent %}

9.5.3. /profiles directory

9.5.3.1. models.py

from django.db import models

class IPAddress(models.Model):
    addr = models.IPAddressField()

    def __unicode__(self):
        return self.addr
class Plugin(models.Model):
    name = models.CharField(max_length=75)
    description = models.CharField(max_length=75)
    filename = models.CharField(max_length=50)

    def __unicode__(self):
        return self.name

class Browser(models.Model):
    name = models.CharField(max_length=75)
    charset = models.CharField(max_length=75)
    encoding = models.CharField(max_length=25)
    language = models.CharField(max_length=25)

    def __unicode__(self):
        return self.name

class OperatingSystem(models.Model):
    name = models.CharField(max_length=100)
    ip_addr = models.ForeignKey(IPAddress)
    last_activity = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return '%s - %s' % (self.ip_addr, self.name)
class UserAgent(models.Model):
    name = models.CharField(max_length=150)

class Profile(models.Model):
    name = models.CharField(max_length=24)
    ipaddrs = models.ManyToManyField(IPAddress, related_name="profile_ips")
    dnsaddrs = models.ManyToManyField(IPAddress, related_name="profile_dns")
    useragent = models.ManyToManyField(UserAgent, related_name="profile_useragent")
    browsers = models.ManyToManyField(Browser, related_name="profile_browser")
    os = models.ManyToManyField(OperatingSystem, related_name="profile_os")
    js_fonts = models.TextField()
    flash_fonts = models.TextField()
    temp = models.BooleanField()
    latitude = models.FloatField(null=True)
    longitude = models.FloatField(null=True)

    ## TODO: Move this to browser
    plugins = models.TextField()

    last_client_ip = models.ForeignKey(IPAddress, null=True)
    last_client_os = models.ForeignKey(OperatingSystem, null=True)
    last_activity = models.DateTimeField(auto_now=True)
    created = models.DateTimeField(auto_now_add=True)
9.5.4. /static directory

In addition to the code below, this directory several static files. ‘flashlogo.swf’ is a Flash file containing the ActionScript code listed above, ‘small.png’ is an image file that contains a single white pixel, ‘fontdetect.js’ is publicly available font detection code [23], ‘font_list.txt’ is a text file containing a list of fonts to check for in JavaScript, and ‘swfobject.js’ is a library for dynamically embedding Flash objects [36].

9.5.4.1. flashlso.js

```javascript
var flash_vars = {
    profileval: "",
    moviename: "flashlogo"
};
```

```javascript
function requestLSOValue() {
    document[flash_vars.moviename].requestLSOValue();
}
```

```javascript
function updateLSOValue( value ) {
    flash_vars.profileval = value;
}
```
function setLSOValue( value ) { 
    document[flash_vars.moviename].setLSOValue( value );
}

9.5.4.2. plugindetect.js

function DetectPlugins() { 
    var plugins = [];

    for( var i = 0; navigator.plugins[i]; ++i ) {
        plugins.push(navigator.plugins[i].filename);
    }

    return plugins;
}

9.5.4.3. style.css

@charset "UTF-8";

* {
    margin:0;
    padding:0;
}
body{
    font:"Comic Sans";
text-align: left;
background-color: #CCC;
background-repeat: repeat;
}

#thewholepage{
    width: 1000px;
    min-height: 500px;
    margin: 0 auto;
    background-color: #FFF;
}

#lefthead{
    width: 300px;
    float: left;
}

#righthead{
    width: 300px;
    text-align: right;
    float: right;
}

#flash{
    width: 100%;
    height: 100%;
}

#smallguy{
width: 1px;
height: 1px;
}