

PhishBouncer: An HTTPS proxy for attribute-based prevention of Phishing Attacks

ABSTRACT

This paper describes an innovative approach toward defending against phishing attacks by using HTTPS proxying and attribute-based checks. After a short overview of phishing, we describe the functional architecture of the PhishBouncer HTTPS proxy together with various deployment options. We then explain a number of anti-phishing algorithms implemented as plugins and highlight which attributes of phishing sites they consider. Next, we describe in detail how the proxy intercepts SSL traffic for HTTPS proxying. To assess the effectiveness and applicability of this prototype, we performed extensive experimental testing. We present a fully automated crawling framework that we developed for testing, along with the main experimental results.

1. INTRODUCTION

Over the past decade, online identity fraud has transformed from being a small scale criminal activity of computer geeks to a widespread phenomenon costing billions of dollars in damages each year.

Cyber criminals use phishing predominantly as a technique for obtaining identity-related information, such as social security numbers or bank account numbers. In a typical phishing scenario, a cyber criminal sets up a fake web site that looks similar to the login page of a target financial institution and sends out a massive amount of email to trick people into logging into the fake web site and entering personal information. The cost incurred by criminals is very low and within a short period of time they can successfully complete an attack cycle and hide their tracks. These facts have fueled the phenomenal growth of phishing attacks[23].

A vast majority of existing anti-phishing products follow a signature-based approach. Users, ISPs, and security staff of financial companies provide suspected URLs as input to a centralized blacklist service, which disseminates vetted blacklists to end clients (mostly browser extensions) for enforcement. This approach ties the effectiveness of anti-phishing products to the accuracy and timeliness of signa-

ture updates. It is impractical to assume that all active phishing sites will be reported to the centralized service on time. Vetting of the reported sites adds some amount of time delay. Therefore, attackers always have a window of opportunity during which a significant fraction of end clients operate without the protection of updated signatures.

Another common aspect of existing products is the insertion point of the anti-phishing technology. Most existing technologies are inserted directly into web browsers via plugin frameworks (such as Browser Helper Objects [26]). Although plugins are good for providing transparent insertion, they are susceptible to buffer overflow attacks on end systems and browsers. Malware frequently gets unwillingly downloaded by users and executed on their computers. Due to poor support in current OSES for strong process isolation, such malware can corrupt the users' web browser and simply disable anti-phishing plugins, install key loggers to record and exfiltrate information, or install root kits to evade host-based intrusion detection systems. Another negative aspect of browser plugins is that custom browser-specific code is required for different browser platforms, which limits support to Microsoft Internet Explorer and Firefox in most cases. Furthermore, the plugins consume additional CPU, disk space, and memory resources on the host running the browser, limiting deployment on small embedded devices such as internet-ready cell phones.

We have developed a different anti-phishing approach in PhishBouncer which uses attribute-based checks to implement both reactive and proactive anti-phishing defenses. These checks do not require signature creation and are strategically placed into the client-server communication pathway via an HTTPS proxy.

We claim that by focusing on common attributes of phishing attacks (rather than specific signatures) and user behavior over time, the PhishBouncer approach doesn't require timely updates and therefore doesn't suffer the problems associated with them. Since attribute checks operate on a wide variety of phishing attack instances by looking at generic features and gathering data autonomously if needed, we are able to significantly reduce the amount of pushed content to the end systems and thereby reduce a large part of the operational cost of anti-phishing services. Furthermore, PhishBouncer provides enhanced protection against previously unknown phishing attacks as in most cases some common attributes stay invariant.

Placement of the anti-phishing checks in an HTTPS proxy provides stronger isolation guarantees and increased flexibility compared to browser plugins. Since the proxy is a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

APWG eCrime Researchers Summit '07 Carnegie Mellon Cylab, Pittsburgh, PA, USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

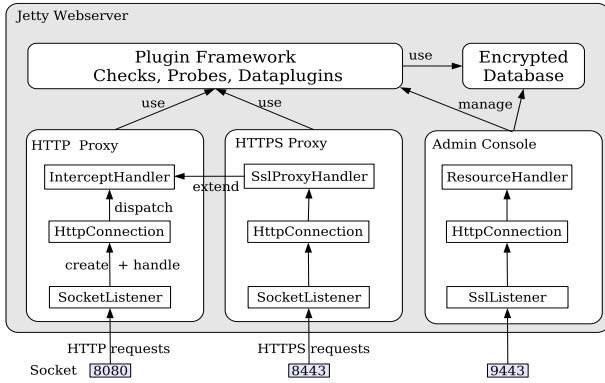


Figure 1: Functional architecture of the PhishBouncer proxy.

dedicated process, it can be protected against direct attacks via technologies that implement process protection domain, such as SELinux[19] or Cisco Security Agent[3]. Even in the absence of supplemental enforcement, the separation of the proxy process from the browser provides a stronger defense against memory corruption attacks than browser plugins such as toolbars. The process proxy can also be deployed on users’ computers, embedded wireless routers, or ISP servers without any changes to the code.

In summary, PhishBouncer’s superior anti-phishing capabilities stem from the following key features:

- Implemented in Java, therefore less vulnerable to traditional exploits (e.g., buffer overflow attacks)
- Architectural solution with stronger guarantees than browser plugins (can catch phishes even if the browser is closed or not part of the communication)
- Browser independent - supports all web browsers
- Operating system independent - supports all operating systems that can run Java
- Highly customizable deployment options - runs on user hosts, wireless routers, or network server
- Open framework and plugin architecture - allows easy addition of new checks
- Attribute-based detection - provides protection against unknown phishing attacks
- Supports reactive and proactive anti-phishing checks
- Supports HTTP and HTTPS

The rest of the paper is organized as follows: Section 2 describes the PhishBouncer architecture together with implementation details for HTTPS proxying, anti-phishing checks and the plugin framework. Section 3 summarizes experimental results, section 4 describes related work and section 5 concludes the paper with a brief description of future work.

2. PHISHBOUNCER ARCHITECTURE

A key aspect of the PhishBouncer approach is its proxy-based architecture. Figure 1 illustrates the design of this proxy, which consists of 4 main modules that are implemented on top of Jetty[18], a popular open-source web server written in Java.

The *Plugin Framework* module provides a means for integration of custom reactive and proactive behavior. All anti-phishing logic is in fact implemented as a set of plugins. We divide plugins into three broad classes based on their role in the overall control flow and threading logic.

The InterceptHandler calls all *Dataplugins* on every HTTP requests and associated response to analyze header and payload. Dataplugins are quite general in nature and not necessarily tied to phishing prevention. For example, a proxy with all checks and probes disabled but the HTTP payload and header dataplugins enabled can be used to record all traffic. We used traffic recorded this way to create a baseline for validating anti-phishing solutions. After performing some data analysis or extraction (for instance computing image hashes on responses), dataplugins may write their result into an in-memory *Database*. Checks frequently query this database for new information and the database gets persisted to disk in encrypted form at a regular configurable interval.

Checks execute sequentially on HTTP requests initiated by the end system’s web browser and decide whether to a) accept the request without further checks (i.e., URL is white listed) b) reject the request without further checks (i.e., URL is blacklisted), or c) set a numeric value $0 < w < 100$ together with a typed choice (acceptPref, rejectPref) to indicate the confidence and choice selection. While the InterceptHandler exits for cases a) and b), it continues to loop through all available checks in case c) before sending all check preferences into an aggregation plugin¹.

In contrast to Checks and Dataplugins which only execute reactively triggered by web browser requests, *Probes* allow us to embed proactive behavior into the PhishBouncer proxy (also referred to as Pb proxy in this paper). Probes contain dedicated threads that trigger monitoring functions at regular configurable intervals. Since phishing relies on reactive human behavior (such as clicking on URLs embedded in phishing emails), probes, being proactive, are not susceptible to those attacks. By registering sensitive information about frequently visited sites (registered sites), PhishBouncer’s probes can monitor valid changes in web sites over time with a high degree of fidelity, including changes in IP addresses and image hashes. The resulting data is then later used by checks in deciding whether to block user requests or not.

The lower part of Figure 1 displays the three access paths into the Pb proxy. The *HTTP Proxy* listens on a configurable network port (e.g., 8080) for incoming HTTP requests, and dispatches the requests to a main handler (InterceptHandler), which in turn makes strategic use of the plugins. This flow is similar in the case of the *HTTPS Proxy*, except that it listens on a different network port (8443) and uses a custom extension of the InterceptHandler (called SslProxyHandler) that contains logic to enable HTTPS interception during a HTTPS Connect request. The third access path is for management of the proxy through an administration console. Management functions include changing order

¹More details on this can be found at the end of section 2.1.

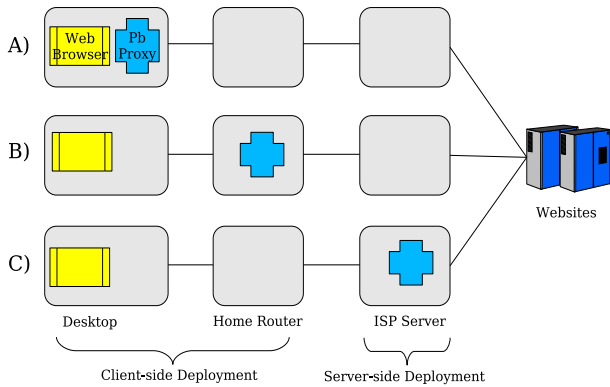


Figure 2: The PhishBouncer proxy can be deployed in various configurations.

of checks and their respective importance weights as well as customization via user-specific data.

The Pb proxy supports deployment on a wide variety of platforms. Depending on available resources and desired service model, the proxy can be hosted at various locations between the end client and the target web site. Figure 2 displays three different deployment scenarios.

In case A), the proxy is co-hosted with the web browser on the end user’s computer. Although this negatively impacts CPU, memory, and disk resources on the end system, this scenario has the benefit of putting the proxy under direct control of the end users. We found that end users feel uncomfortable with disclosing personal sensitive data to external parties, but are more amenable to providing this information to local components as long as it doesn’t leave their machine.

Since many end-users own either a wireless or DSL router and these devices already ship with web server capabilities, we investigated deploying the Pb proxy on a Linksys WRT54G wireless router[7] running OpenWrt[9]. Option B) in Figure 2 shows that the Pb Proxy (represented by the cross symbol) running on a home router. The benefits of this deployment option are increased security through stronger isolation from a potentially virus infected desktop and new marketing opportunities for wireless router manufacturers who could include the Pb proxy as a value added offering. On the downside, the very limited CPU and memory resources of the wireless router significantly lowers the performance of the proxy and would necessitate re-implementation in C++ to be successfully offered as a product.

Deployment option C) places the phishbouncer proxy onto a server platform that resides for instance at an ISP location and is capable of handling hundreds of users interactions concurrently. Such a server-side deployment has the clear benefit of supporting anti-phishing checks for end systems that can only host minimal software (such as cell phones), and it would allow ISPs to offer anti-phishing as a value added service. The major technical challenge of this deployment is to increase PhishBouncer’s scalability to handle increased request load.

As of the writing of this paper, we have mostly used and tested PhishBouncer as deployed on various Linux and Windows end systems (case A)), but have studied the impact of changing over to cases B) and C).

The remainder of this section describes in more detail

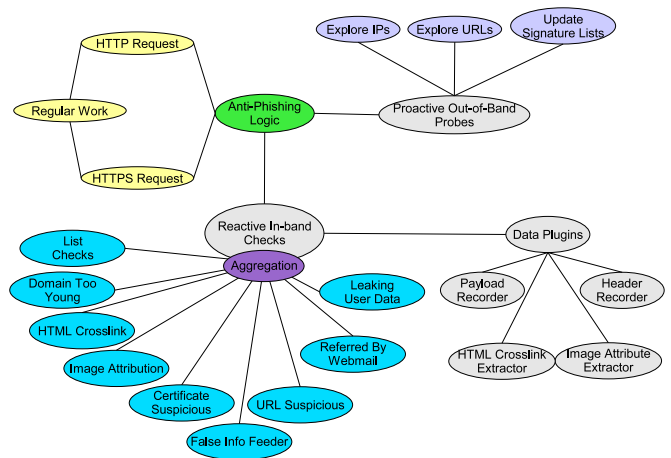


Figure 3: Use case diagram showing all currently implemented checks, probes, and dataplugins.

PhishBouncer’s unique attribute-based anti-phishing functionality, HTTPS Proxying, and extensible plugin framework.

2.1 Anti-Phishing Functionality

This section describes some of the anti-phishing behavior performed by the currently existing set of checks, dataplugins, and probes. Figure 3 displays a use case diagram for the PhishBouncer proxy. As part of performing regular work, a user may visit web sites and therefore generate HTTP and HTTPS requests. These requests are intercepted by the proxy and subjected to anti-phishing logic, which we categorize into reactive checks, proactive probes, and dataplugins as previously explained in section 2.

The following checks and their supporting dataplugins are unique to PhishBouncer, although some variance of the same ideas may have been explored independently by others:

- Suspected By ImageAttribution
- Domain Too Young
- HTML Crosslink
- False Info Feeder
- URL Suspicious
- Certificate Suspicious

The *Suspected by ImageAttribution* check and its associated data plugins look for image similarities between sites the user has previously registered with PhishBouncer and sites the user is trying to go to. If the images are similar but the domains are different, it is an indication that the user is trying to go to a phishing site that either displays back a local copy of the image or links to the real image. PhishBouncer keeps hashes of images for all registered sites and implements a hash-based comparison algorithm.

The idea behind the *Domain Too Young* check is to flag sites which are hosted on very young DNS domains, as phishers frequently create mirror domains such as `bankofbar1.com` instead of `bankofbar.com` and move on to new domains (`bankofbar11.com`) as currently active ones get taken down by ISPs. During the Domain Too Young check, PhishBouncer

performs a WHOIS lookup for outgoing TCP connections to DNS names and IP addresses. After retrieving the WHOIS record, it checks for the age of the domain name and prompts for user feedback if the age falls below a certain threshold. The WHOIS entries are cached locally in PhishBouncer to increase performance. One of the biggest challenges we encountered during implementation of this check is that there is no standardized protocol for finding the correct WHOIS server for a given domain name. We implemented heuristics and mappings similar to the WHOIS Unix command, which mitigated some of WHOIS server lookup problems. The next problem had to do with reliable parsing and data-extraction of creation time data returned by various WHOIS servers around the world. Since WHOIS requests are made in line with HTTP requests, the overhead of WHOIS lookups directly adds to the overall round trip HTTP request latency. We decided on a relatively small timeout of 200ms for timing out WHOIS requests, which may cut out a large number of slow WHOIS servers.

The *HTML Crosslink* check looks at responses from non-registered sites and counts the number of links the page has to any of the registered sites (in particular image cross-links). A high number of cross-links is indicative of a phishing site.

The key idea behind the *False Info Feeder* check is that a phishing site is likely to indiscriminately accept user supplied data, since it has no way of knowing whether the user supplied data is valid or not. This can be used to detect the presence of phishing sites. For instance, PhishBouncer may decide to send a bogus user name and password combination to a site that has already been declared suspicious by previous checks. If the site does not reply with an error, it further increases the sites phishyness factor².

The *Certificate Suspicious* check validates site certificates presented during SSL handshake and extends the typical usage by looking for Certification Authority (CA) consistency over time. The basic idea of this check is based on the observation that companies rarely change the CA used for signing their site certificates. We can use this fact to flag sites that present certificates signed by a different CA than expected. An associated probe proactively monitors registered sites to keep the set of expected CAs current.

The Certificate Suspicious check tries to assert the following conditions: 1) Certificate validity - certificate timestamp is in current time range. 2) Trusted CA - certificate is signed by a trusted CA. Currently, the list of the trusted CA is imported from that of the JRE. 3) Host Name of the server equals the CN field of the certificate. 4) The CA's DN field in the certificate matches that of the history data base. 5) Crypto signature algorithms run error-free.

For conditions 1 and 5, traffic to the site is automatically blocked when the check reports a problem, since they are clear signs of attack. For conditions 2, 3, and 4, the user will be prompted to either accept this certificate or to reject the certificate. Once the certificate is accepted, this certificate will be automatically accepted for that site until PhishBouncer gets restarted to increase user satisfaction. Likewise, once the certificate is rejected, it will be rejected

²As an interesting extension, PhishBouncer could also send fake credit card numbers by implementing a credit card number generator that not only passes the rough number check but also guarantees that the number generated is not valid.

until restart.

In addition to these checks, we have implemented many previously known and published algorithms as plugins, including the *URL Suspicious* check which tries to identify phishing sites by looking at the characteristics of the URL they present (puny-encoding [12], edit distance); the *Leaking User Data* check which searches through outgoing traffic for sensitive user information such as social security numbers; the *Phish Signature Match* check which compares URLs against a phishing signature feed; and the *Referred By Web-mail* check which looks at the Referer header of HTTP requests and declares requests suspect (with low confidence) if they originate from a web-based email provider.

If an HTTP request emerges from the chain of checks together with a vector of check results, it will enter the Aggregation plugin. Aggregation of multiple check results plays an important role as we use it to increase true positives and decrease false positive rates via a multi-factor weight function. PhishBouncer combines the outcome of individual checks in such a way that multiple weak suspicions have a multiplicative effect.

We use the following aggregation formula for computing the aggregate value V :

$$V = \sum_{t=1}^{t=3} O_t * T_t(w_{1..i}) \quad (1)$$

Each term T_t is an aggregation term over the distributions of i fired checks, where T_1 averages over single weights, T_2 averages over pair-wise combinations, and T_3 averages over three way combinations. O_t is a term weight between 0 and 1 specified via the administration console, and $0 < w_i < 100$ is the confidence value of an individual check with $w_i = 99$ expressing near certainty that the request goes to a phishing site.

In most of our experiments, we set up O_t to a value such that a single check with a high value of $w_i = 100$ does not cause a rejection of the request. We further calibrated the system to start rejecting requests as soon as two checks fire, given that their respective values $w_i, w_j > 1$.

It turns out that empirically determining a good set of w_i and O_t is non-trivial. It is probably better to define rules that take into account the number of fired checks or learn accurate values via statistical machine learning during off line training.

2.2 HTTPS Proxying

Insertion of the proxy into the non-encrypted HTTP client-server path is straightforward and is condensed down to changing the client's HTTP web browser's proxy settings during phishbouncer installation. To prevent an attacker from replacing the proxy setting to a proxy of his own, firewall rules should be set to only allow outgoing web traffic through the Pb proxy.

Insertion becomes significantly more difficult for encrypted traffic. For intercepting encrypted HTTPS requests, the browser's proxy settings are changed accordingly to redirect requests to PhishBouncer's HTTPS proxy port. That said, this only allows for encrypted requests to flow into the Pb proxy, as the underlying SSL protocol is specifically designed to prevent man-in-the-middle use cases, whether the man is benign or not. So how were we able to intercept HTTPS requests? The answer lies in the way trust relationships are established.

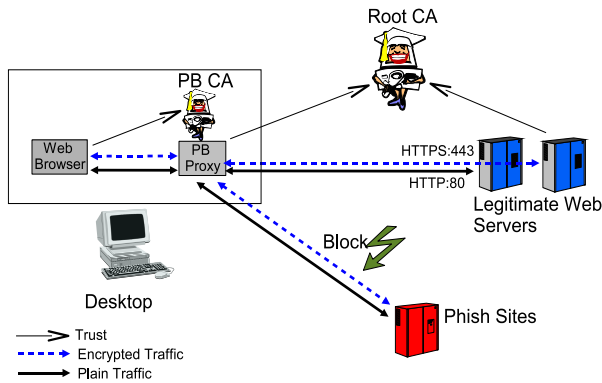


Figure 4: PhishBouncer Proxy Acting as Trusted Man-In-The-Middle

Figure 4 displays the architectural layout³ for proxying of encrypted HTTPS traffic.

In a regular use case without any HTTPS proxy, SSL relies on a PKI infrastructure for connection establishment [25]. Following a general description of the SSL protocol, the client issues a connection request to the server, which the server acknowledges with a response containing a certificate signed by a CA. The client then continues to perform a set of checks on the server certificate, the main one of which is to verify that the CA’s signature is valid. The SSL transactions essentially establish a unidirectional trust relationship between the browser and the target web server via a commonly trusted CA (small black line).

With the Pb proxy in the mix, the protocol becomes a little more complex. The Pb proxy takes on the role of a server when communicating with the browser, and the role of a browser when communicating with the target web server. This requires the Pb proxy to dynamically generate X509 certificates for each web site it is proxying⁴. Since the certificate generation process via CAs typically requires off line identity verification, the Pb proxy hosts a second CA (called Pb CA in Figure 4). During installation, the web browser’s settings are configured to trust signatures from the Pb CA⁵. As a result, the overall trust relationship between browser and target web server can now be decomposed into two daisy-chained relationships, one between the browser to the Pb proxy, and a second one between the Pb proxy and the target web server.

Does the Pb proxy introduce additional security vulnerabilities through breaking the end-to-end encryption between browser and web server? The answer to this question depends on the relative trustworthiness of the Pb proxy compared to the browser and target web server and where it is deployed. Consider the extreme case of a highly secure desktop which hosts the browser and a highly secure web server. Deploying the Pb proxy on an ISP server which may co-host other applications and not have the latest security patches installed would significantly lower the overall security of web transactions flowing through it. On the other hand, in a

³For illustration purposes, we assume an end client deployment of the Pb proxy shown as “Pb Proxy”.

⁴To increase generation performance, key pairs can be reused across certificates.

⁵Alternatively, the Pb CA can be signed by a common Root CA.

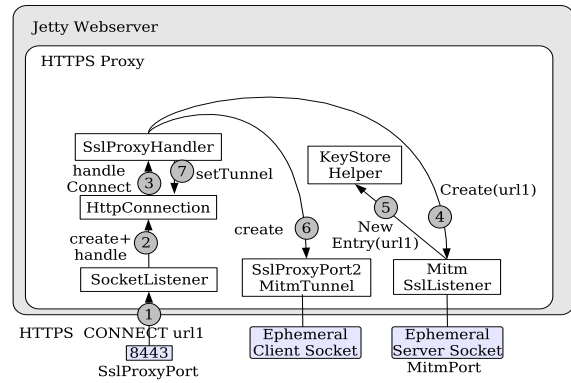


Figure 5: Handling of HTTPS Connect requests.

scenario where Pb proxy is co-located with the web browser on the same desktop, we’d expect it would be more difficult for attackers to subvert the Java-based stand alone Pb proxy process (which only listens on localhost) compared to a C++ web browser running Javascript. In both cases, data is never sent unencrypted over the network, so the guarantees provided by SSL are not affected.

The remaining part of this section describes in more detail the control and data flows for HTTPS CONNECT and data requests within the Jetty web server. Figure 5 shows the call sequence for establishing the instances that implement the SSL man-in-the-middle interception. Upon receiving an HTTPS CONNECT request through its SslProxyPort (1) for a specific URL (url1), the SocketListener creates an associated HttpConnection (or looks up an previously established one) in step (2) and forwards the request to the SslProxyHandler (3). It checks whether a tunnel has already been created for the target web site, and if not will start the process of establishing one by creating a new SSL Server Socket and associated MitmSslListener (4) for url1. The function of the MitmSslListener is to project the SSL identity of the target web site’s SSL listener back to the client’s web browser. For this reason, it will generate a new site certificate via the KeyStoreHelper (5) for url1 signed by the PhishBouncer CA and bind it to the SSL Server Socket listening on an ephemeral port (MitmPort). After establishing a local SSL endpoint for url1, the SslProxyHandler creates a new client socket (TunnelClientPort) and creates tunnel (SslProxyPort2MitmTunnel) which simply forwards all data send into TunnelClientPort to MitmPort (6). Finally, the SslProxyHandler tells the HttpConnection to use the newly created tunnel for all further communication over this connection (7).

Figure 6 displays the data flow over an existing tunnel. Incoming encrypted data requests (such as HTTP GET) are dispatched to the HttpConnection (via 1,2) which sends them via the MitmTunnel (3,4) to the MitmListener. The MitmListener decrypts the HTTP request and then forwards the request to the InterceptHandler (5,6) which in turn forwards the request to the various plugins. In summary, the MitmListener’s main job is to perform the decryption and encryption operation, while the responsibility of the other classes lies in dispatching the data to the right places in a thread-safe manner.

2.3 Plugin Framework

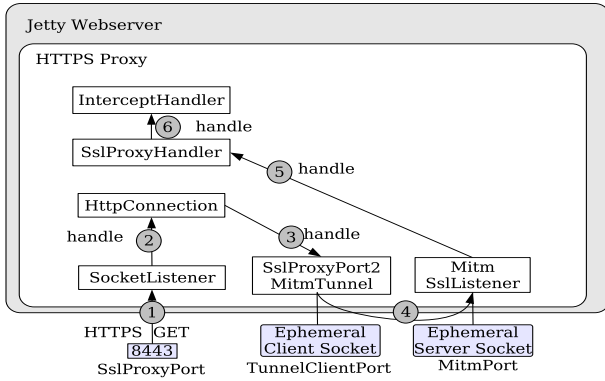


Figure 6: Handling of HTTPS Data requests.

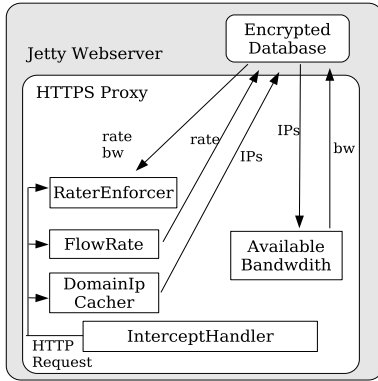


Figure 7: It's easy to extend PhishBouncer by encapsulating auto-adaptive logic as plugins.

We realized early on that incorporating flexibility into an anti-phishing solution is crucial for its effectiveness. We therefore designed the plugin framework in such a way that the security community can add in new functionality in a straightforward way and leverage the rather complex plumbing code for HTTPS interception and plugin configuration via a management console.

To illustrate how to use the framework, this section describes an example of how to integrate application-level rate limiting on outgoing HTTP requests. The desired behavior is to not to exceed $rate1$ number of HTTP requests per second to domain d if the available bandwidth to d is 80 MBPS, while not to exceed $rate2$ number of HTTP requests per second if the available bandwidth is less than 10 MBPS, with $rate1 < rate2$.

As displayed in Figure 7, we decompose the implementation of this logic into four plugins.

The *RateEnforcer* makes the decision of whether to forward HTTP requests or start rate limiting of their frequency exceeds a customizable threshold. In essence, this check is the embodiment of the rules described earlier, which will start dropping requests if the $rate > rate1$ or $rate > rate2$.

The *Flowrate* plugin's task is to compute the average flow rate of HTTP requests ($rate$), which is a simple matter of updating a mean value every time it gets a new HTTP request. It then stores $rate$ in the encrypted database, which the *RateEnforcer* queries when evaluating its decision rule.

The *DomainIpCacher* keeps track of the live IP addresses

that currently serve a given domain. Every time a HTTP request with a DNS name requests flows through this plugin, it performs a DNS resolution and adds the IP address to a DNS mapping hash table stored in the encrypted database. To limit the growth of this hash table, the plugin will remove entries after a configurable amount of time.

While the previous described plugins are reactively executed in line with HTTP requests, the *AvailableBandwidth* probe runs at regular intervals to actively measure the network bandwidth between two configurable endpoints. Since we are interested in determining the available bandwidth to a given domain and a domain may be hosted on multiple servers, the probe needs to know which IP endpoints it should consider during its measurements. It gets this information by looking up IP addresses from the DNS mapping hash table that is maintained by the *DomainIpCacher* plugin. This probe may then perform round-trip time measurements (e.g. via ping) to all IP addresses for a given domain, compute an average available bandwidth rate, and store the it in the encrypted database for use by the *RateEnforcer*.

As this example shows, the plugin framework enables modular implementation of non-security related QoS adaptive behavior. We envision plugins to implement small specialized algorithms that can be integrated together simply by using the data they provide to the encrypted database. Following our previous work in QoS adaptive middleware through the Quality Objects [20] and CORBA component models [22], we intend to extend the plugin framework to provide language support expressing QoS adaptive logic as future work.

3. EXPERIMENTAL VALIDATION

There is currently no anti-phishing benchmark or standardized set of data for cross-product evaluation. Most of the claims made by vendors of available products are based on proprietary test data and testing methodology. In contrast, we developed a test framework which can evaluate a generic anti-phishing technology against the latest live phishing sites⁶ and used this framework to evaluate the effectiveness of the Pb proxy. Details of this experimentation framework and some preliminary findings are presented in this section.

We also subjected the Pb proxy to experimental validation by independent testers and limited field testing. Findings from these tests are not ready for reporting and therefore not included in this paper (except for anecdotal references).

3.1 Experimentation Setup

Figure 8 displays the main component used in the experimentation process together with an experimentation work flow.

In the first step, the automated driver obtains the latest set of URLs from a list service such as the APWG phish feed and builds a differential to the previously downloaded set so that only unseen URLs are processed.

For each new URL_X , the driver proceeds to start up a PhishBouncer proxy with all anti-phishing checks disabled and maximum logging enabled in step 2 ("Pb Proxy RecordOnly") in Figure 8). Next, the driver starts a Firefox web browser configured to use the recordonly proxy and instructs

⁶We currently use the feed from APWG, but the framework can be plugged into other feeds as well.

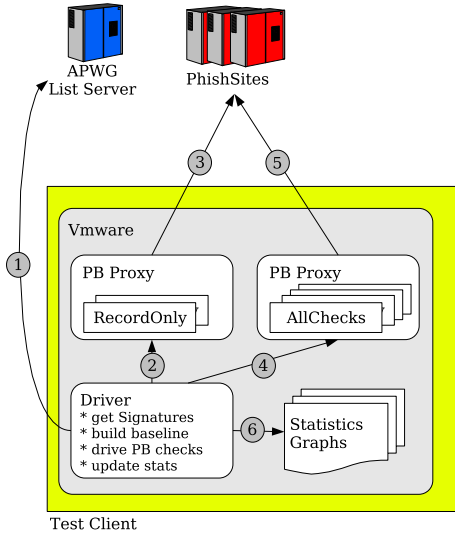


Figure 8: Experimental setup for validating anti-phishing technologies.

Firefox to display the phish URL URL_X . The proxy records all headers and payloads (including Javascript code segments, images, HTTP headers, SSL handshakes) together with timing information for post-processing (3).

After building the baseline, the driver starts a proxy instance with all checks enabled (4), and again starts an associated Firefox instance to display URL_X (5). This time, multiple checks may indicate reject preferences for requests and all check results are logged. Note that is presently a limitation of this test harness that it cannot generate requests that simulate human user’s exploration of a web page and submission of data, e.g. a user logging into an account. This implies that checks that are based on user specific browsing pattern and data are not exercised in this test harness.

To conclude the cycle, the driver performs post-processing to extract metrics from the set of gathered log files and filter out invalid URLs. A URL is filtered if it a) is not live anymore because the phish site has been taken down already (resulting in 4XX and 5xx error codes) b) causes errors because other sites it depends on have changed c) causes errors induced by the archiving framework. The filtering is solely performed on the basis of log files from the recordonly proxy instance in order to minimize any impact of the anti-phishing checks on the selection process. The driver then proceeds to extract important statistics from the proxy log files, including check scores and results, processing latencies, and interesting check-specific metrics, e.g., the domain age. Finally, the driver calls the R[21] statistics package to group data, perform statistics, and plot results.

The next section describes the main results and conclusions on the effectiveness of PhishBouncer’s anti-phishing checks.

3.2 Preliminary Results

We gathered the data set of phishing sites used in this paper by running the automated crawling framework described in the previous section in a non-continuous way over the period from Feb 20 2007 until Feb 23 2007 with PhishBouncer version 2.0. During that period, we actively crawled 1436 re-

ported phishing URLs. Surprisingly, only 224 (15.6%) valid crawling runs remained after filtering out sites that caused errors. We believe that this number can be explained by the fact that most companies have a policy of only report a phishing URL to the APWG list service **after** the site has been taken down. Another potential explanation could be that the phishers start blocking access to their sites for certain source IP addresses that have been identified to run crawlers.

The following subsections summarize our claims about PhishBouncer’s effectiveness based on analysis of the 224 error-free runs.

3.2.1 True Positive Rate

Figure 9 displays the check results as a bar plot. The y axis displays the percentage of runs (between 0 and 100), where a run consists of visiting a phishing URL. The x axis divides the outcome of a check into 4 categories: *accept* (*reject*) if a check clearly accepted (rejected) the URL without further checks and *accept_pref* (*reject_pref*) if a check didn’t (did) declare the URL suspicious. Each bar in a column represents a single check, and the bars within a column are ordered from left to right in correspondence with a top-down read of the check names from the agenda.

Since a single run may cause a large number of HTTP requests, this diagram aggregates the outcomes of individual requests as follows. We count a run in the *accept_pref* (*reject_pref*) column of a check if that check flagged at least one request as *accept_pref* (*reject_pref*). Note that a run may be counted in both columns as shown for the URL Suspicious check. We perform a similar computation for the *reject* column of a check in that a run gets counted as *rejected* if the check rejected at least one request from that run. Because the Pb proxy terminates the run as soon as it hits a *reject*, we further compute the *accept* percentage as the total number of runs - the number of rejects.

The *accept* and *reject* columns are only populated by the Aggregator checks, which tells us that none of the definite checks (such as Certificate Suspicious or URL Blacklisted) fired. Furthermore, the Aggregator bars indicate that 65% of the URLs were rejected by the Aggregator.

The *accept_pref* and *reject_pref* columns allow us to drill down and decompose the aggregated decision into individual base check results. The main reason for rejection of URLs seems to stem from *reject_pref* values from the URL Suspicious and HTML Crosslink checks, together with a small number contributed by the DomainTooYoung Suspected By ImageAttribution checks.

The False Info Feeder, Leaking User Data, and Leaking Registered Sites Data checks operate on HTTP POST requests that submit user data to remote websites. As pointed out earlier, we currently don’t simulate such behavior in the automated test framework, which is why these checks only have *accept_pref* bars.

None of the phishing URLs were HTTPS based. However, some caused generation of HTTPS requests to display images, which explain a small *accept_pref* bar for the Certificate Suspicious check.

A number of comment about these observations are in order. First, we realize that the size of the data set described in this paper is small and more extensive testing is needed to establish stronger confidence on the effectiveness claims of attribute checks. Furthermore, since many of the reported

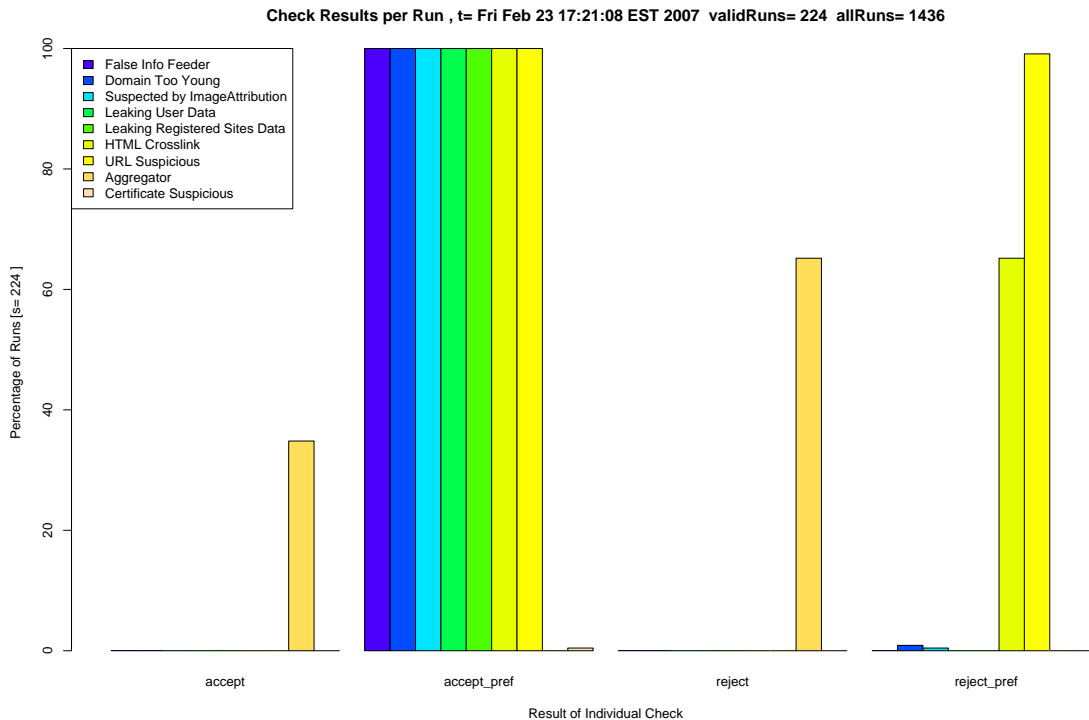


Figure 9: Distribution of check results as a fraction of runs.

phishing URLs from the APWG list were not live, the remaining live sites may represent a skewed distribution and might not be representative of the set of unknown phishing sites that have not been reported yet. Another interesting observation is that the APWG list goes through spurs of URL permutations such that in a short amount of time, a very large amount of similar looking URLs may get reported, e.g., <http://www.foobar1.com>, <http://www.foobar2.com> etc. This may skew the results in Figure 9 to favor checks that are better at catching URL permutation attacks over other checks. Last, the weights on the individual checks and the aggregation formula have a large impact on this result. Although we did not calibrate those parameters directly for the dataset at hand, it is unclear whether they need to be adjusted for future datasets and how to best optimize them for the current dataset.

3.2.2 False Positive Rate

For determining the false positive rate, we utilized the previously described experimentation setup, except that we replaced the APWG signature feed with a static list of 23 commonly used commercial bank and news sites. The results were surprisingly good in that PhishBouncer didn't reject any of the 23 URLs. That said, we received multiple incident reports from users indicating that PhishBouncer had blocked access to legitimate sites. In one instance, a user was reading email from his web-based email provider and clicked on a link to a trusted commercial site which PhishBouncer blocked without giving the user the option of a manual override. The lesson learned here is that users are very sensitive to false positives and tend to disable the Pb proxy altogether upon encountering very few false positives.

Our plans are to extend the false positive test coverage to a larger more representative set of white-listed websites and domains and to allow manual override of **all** block decisions while adding a reporting mechanism for gathering and evaluating override occurrences.

3.2.3 Performance Overhead

Performance metrics are important for any cyber-defense technology in the sense that technologies which introduce a clearly noticeable delay face increased resistance to adoption by user communities. To minimize performance impact, we implemented the Pb proxy on top of the high performance Jetty web server and implemented various optimizations in the SSL proxy architecture to keep request latencies within user acceptable levels.

We measured the overhead of the HTTPS interception in a lab environment and through field testing. Experimental data shows that the HTTPS proxying introduces an overhead of roughly 100%, and we contribute much of this overhead to crypto operations and session multiplexing performed in Java. Even in the current form, the delay introduced by the Pb proxy doesn't noticeably impact the user web surfing experience, as field testing participants did not notice a significant performance penalty. Since we expect the plumbing overhead to stay invariant as we add more complex checks to PhishBouncer, we narrowly focus on the performance overhead introduced by the checks in the remainder of this section.

Figure 10 displays two box plots⁷[24] of round-trip request

⁷Box plots are a compact way of visualizing and comparing distributions, in which the box represents to interquartile range while the horizontal line in the box represents the me-

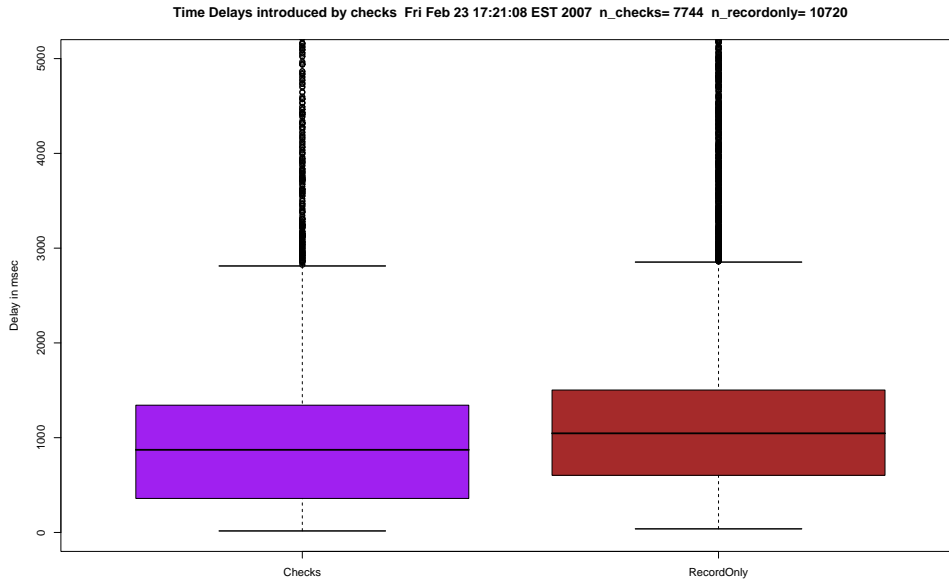


Figure 10: Overhead of PhishBouncer checks - indistinguishable from regular traffic

latencies. The y axis displays the round trip latencies of requests measured by the timestamp difference of requests entering the proxy from the web browser and the corresponding response leaving the proxy to the web browser. On the x axis, the left distribution (Checks) shows the latency distribution for an instance of Pb Proxy that has all checks enabled, while the right distribution (RecordOnly) shows the latencies when all checks are disabled.

Overall, the two distributions are not significantly different as their interquartile ranges overlap to a large extent (from 200 to 1500 ms) and both distributions have a large number of outliers (some even > 50000 ms). We suspect that available network bandwidth to the phishing sites together with available CPU resources of those sites have the biggest impact on round trip latencies. Since these resource metrics are invariant to the introduction of PhishBouncer’s checks, this would explain the similarity between the two distributions.

It is worth pointing out that the median for the “Checks” distribution is slightly smaller compared to the “RecordOnly” median. One possible explanation for this could be that PhishBouncer blocks a significant number of requests with larger latencies (such as requests for images, videos), which seems to be supported by a difference in the distribution sizes ($n_checks = 7744 < n_recordonly = 10720$).

4. RELATED WORK

Various commercial products and research prototypes in the areas of anti-phishing technologies, HTTPS interception, and automated crawling frameworks have co-evolved since we first started working on PhishBouncer in late 2004. Rather than focusing on specific instances, our goal in this section is to describe broad categories of solutions and their attributes together with some example solutions.

Signature-based anti-phishing products are widely deployed in the commercial market, with examples such as Netcraft[8] or the Google Safe Browsing blacklist[5]. Although PhishBouncer provides capabilities to interface with existing signature services through plugins, its attribute-based checks allow for prevention of phishing attacks in the absence of any signatures. Behavioral anti-phishing tools, such as the SpoofGuard[17] research prototype or the Deepnet browser extension[4], have significant overlap with PhishBouncer’s attribute-based checks but differ on other attributes such as the degree of experimental validation and choice of injection point (browser plugin vs. HTTPS proxy). Behavior-based checks are also found in commercial products like Web Caller-ID[15]; other products like Norton Confidential[13] block known phishing websites and warn against suspicious ones. But none of these products are HTTPS proxy based.

Various HTTP and HTTPS proxy implementations exist for debugging purposes (Burp Proxy[1], Charles Proxy[2]) and web filtering (WebCleaner[14], Privoxy[11]). What makes PhishBouncer unique is its use of HTTPS proxying for anti-phishing purposes, while anti-phishing browser plugins, such as Google’s Safe Browsing Firefox plugin[6], Deepnet Antiphishing, or Microsoft Phishing Filter[16], are widely deployed. Compared to PhishBouncer, those browser plugins provide a tighter integration with the end browser gui at the cost of weaker architectural security guarantees.

We were unable to find an integrated framework that would allow us to simply link in the Pb proxy and evaluate its effectiveness in preventing phishing attacks. However, building blocks for constructing such an experimental validation framework are widely available, i.e. web clients, scripting languages, statistics packages. We are aware of Sparta’s ongoing Phisherman[10] effort which automatically extracts phishing signatures from spam email. We anticipate that the Phisherman infrastructure has significant overlap with the plumbing infrastructure of our test harness.

The whiskers visualize the smallest and largest value of the distribution, and circles represent outliers.

5. CONCLUSION

The primary experimentation result shows a 65zero-day attacks using only 4 types of checks is a promising start. This implies that embedding attribute-based anti-phishing checks into an HTTPS proxy could be a viable defense against phishing, complementing signature and block list based defenses. End users, security practitioners, and anti-phishing researchers all can benefit from using Pb proxy. Toward that end, we intend to make the Pb proxy and the automated testing framework available as open-source software.

One direction that we did not explore so far but intend pursue after the open-source release is to transform the Pb proxy into an expert assistant to technicians responsible for vetting reported phish sites. Similarly, the automated crawling and recording framework could also be extended for evaluating diverse anti-phishing technologies against common benchmark tests.

Our extensive testing of the prototype proxy shows that HTTPS interception is feasible in practice. We already started taking advantage of the proxy's built in flexibility to extend its use beyond phishing attack prevention and into developing adaptive web based distributed systems like the example presented in section 2.3. More work remains to be done in this area.

Another direction of future work we would like to pursue involves addressing the issues that arise in server-scale deployments of the Pb proxy. In this situation, a single proxy instance needs to support hundreds of parallel sessions, and the past historical behavior of the users will not be readily available or visible. One interesting question is whether it is possible to boost the aggregate efficiency over all sessions through strategic ordering of checks. How to collect historical behavior patterns of users and build the attribute profile of the sites they visit for the purposes of anti-phishing algorithms without violating the users' privacy rights will be another challenge.

6. REFERENCES

- [1] Burp proxy. <http://www.portswigger.net/proxy/>.
- [2] Charles web debugging proxy. <http://www.xk72.com/charles/>.
- [3] Cisco whitepaper: Cisco security agent-enterprise solution for protection against spyware and adware.
- [4] Deepnet antiphishing. <http://www.deepnettechnologies.com/products/dap.asp>.
- [5] Google blacklist. <http://sb.google.com/safebrowsing/update?version=google-black-url:1:1>.
- [6] Google safe browsing. <http://www.google.com/tools/firefox/safebrowsing/>.
- [7] Linksys wireless routers. <http://www.linksys.com>.
- [8] Netcraft anti-phishing toolbar. <http://toolbar.netcraft.com/>.
- [9] Openwrt - linux distribution for embedded devices. <http://openwrt.org>.
- [10] Phisherman - real-time phishing data collection, validation, dissemination, and archival. <http://www.issosparta.com/documents/phisherman.pdf>.
- [11] Privoxy. <http://www.privoxy.org/>.
- [12] Punycode: A bootstring encoding of unicode for internationalized domain names in applications (idna). <http://tools.ietf.org/html/rfc3492>.
- [13] Symantec norton confidential. http://www.symantec.com/home_homeoffice/products/features.jsp?
- [14] Webcleaner - a filtering http proxy. <http://webcleaner.sourceforge.net/>.
- [15] Wholesecurity web caller-id. http://citrix.wholesecurity.com/documents/brochures/WCID_Brochure.pdf.
- [16] Microsoft anti-phishing white paper. *Anti-phishing_White_Paper.doc*, 2005.
- [17] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell. Client-side defense against web-based identity theft. In *11th Annual Network and Distributed System Security Symposium (NDSS '04)*, 2004.
- [18] Jetty. Jetty homepage. <http://www.mortbay.org/>.
- [19] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference*, 2001.
- [20] J. Loyall, R. Schantz, J. Zinky, and D. Bakken. Specifying and measuring quality of service in distributed object systems. In *IEEE Int'l Symp. Object-Oriented Real-Time Distributed Comp.*, Apr. 1998. Kyoto, Japan.
- [21] The r project homepage. <http://www.r-project.org/>.
- [22] P. Sharma, J. Loyall, R. Schantz, et al. Using composition of qos components to provide dynamic, end-to-end qos in distributed embedded applications - a middleware approach. In *IEEE Internet Computing*, May 2006.
- [23] A. P. A. Trends. Apwg phishing activity trends. http://www.antiphishing.org/reports/apwg_report_february_2007.pdf.
- [24] J. Turkey. *Exploratory Data Analysis*. Addison-Wesley, 1977.
- [25] D. Wagner and B. Schneier. Analysis of the ssl 3.0 protocol. *The Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.
- [26] Wikipedia. Browser helper object. http://en.wikipedia.org/wiki/Browser_Helper_Object.