# A PROXY-BASED APPROACH TO TAKE CRYPTOGRAPHY OUT OF THE BROWSERS FOR BETTER SECURITY AND USABILITY

**Marco Antônio Carnut (kiko@tempest.com.br)**
Tempest Security Technologies

**Evandro Curvelo Hora (evandro@tempest.com.br)**
Universidade Federal de Sergipe - DCCE/CCET

ABSTRACT

*This paper describes the implementation of a multiplatform selective filtering/rewriting HTTP proxy that allows the PKI-related operations – such as digital certificate issuance, web form field signing and HTTPS client authentication – to be performed entirely outside the browser, even though the browser continues to be used for what it's good at: rendering web pages. Implications such as better usability through improved user interfaces are discussed in light of a prototype implementation.*

## 1 INTRODUCTION

The SSL/TLS protocols were originally designed to provide encrypted and authenticated channels for web servers and clients. Even today, they are almost exclusively used to authenticate servers, despite its support for client authentication. There are many reasons for that: in [4], it is shown that getting a client certificate – even a free an instantaneous one – is too much of a hassle for the average user. Internet Explorer (IE), the most popular web browser, makes it all too easy to store the certificate without a passphrase; besides, its client certificate-based logon window is confusing, showing expired and revoked certificates along with valid ones and it is outfitted with a "remember password" checkbox that causes the passphrase to be stored unencrypted, invalidating much of the security the process might provide.

The way failures are handled is also confusing: when the server can't validate the client certificate (either because it couldn't build a trusted certificate chain or the client certificate was found to be revoked), it simply breaks the connection; there are no provisions to redirect the user to a nice page explaining what went wrong and how to fix it.

All these usability problems cause enough user rejection that webmasters find it simpler to use weaker authentication schemes such as name+password+cookies. Although vulnerabilities have been discovered (and in some cases fixed) in most browser's crypto implementations, bad human-computer interface (HCI) is often appointed as a serious hinderance to PKI adoption in general [14] and client-based authentication in particular [18].

There have been a few attempts to improve the user-friendliness of client authentication, such as VeriSign's Personal Trust Agent [17] and RSADSI's Keon WebPassport [16]. However, as they are both ActiveX controls, they are Windows-only solutions and since they are activated after the SSL handshake, they have to resort to proprietary authentication schemes.

Another great promise brought by public key cryptography is the use of digital signatures as a way to detect tampering on digital documents. Some web browsers can natively sign the contents of web form fields, but many – most notably IE – do not support this feature. In IE, it can be implemented using ActiveX or even Java (although that requires installing CAPICOM, making the process less transparent), but they tend to be too cumbersome for large-scale deployment.

This paper investigates an alternative way to provide client certificate-based authentication and web form signature, along necessary subsidiary services such as digital certificate issuance, by performing all the cryptographic and user interface chores in a separate program: we use a selective cryptographic filtering/rewriting HTTP proxy to implement all the PKI-related features, leaving to the browser only what it's good at: rendering web pages. This approach has the advantage that it works with any browser that supports proxies.

Specifically, we wanted to make a general purpose utility for handling digital certificates that provided easy-to-use digital signature generation and verification functions; and that could be integrated with the web browser to allow web form signature and client certificate authentication in HTTPS with a much better user interface and security features under our control. We also wanted this utility to be a testbed for new HCI ideas applied to client-side (primarily, but not limited to, web-based) PKI applications.

This paper focuses on the cryptographic, PKI and protocol issues needed to "take crypto on our own hands" (as opposed to letting the browsers do it), while simultaneously striving to maintain backwards compatibility. Although we do make extensive use of screenshots to illustrate some features and preliminary user interface (UI) ideas we implemented – and sometimes we even indulge in describing some of its details and user feedback we received –, an analysis of the merits of our tool's UI is beyond the scope of this paper, for it requires entirely different approaches and techniques. What we want here is to show one possible way it can be done.

Besides general familiarity with the X.509/PKIX/PKCS standards and PKIs in general, this text assumes the reader has considerable familiarity with the HTTP [1] and HTTPS [2, 3] protocols.

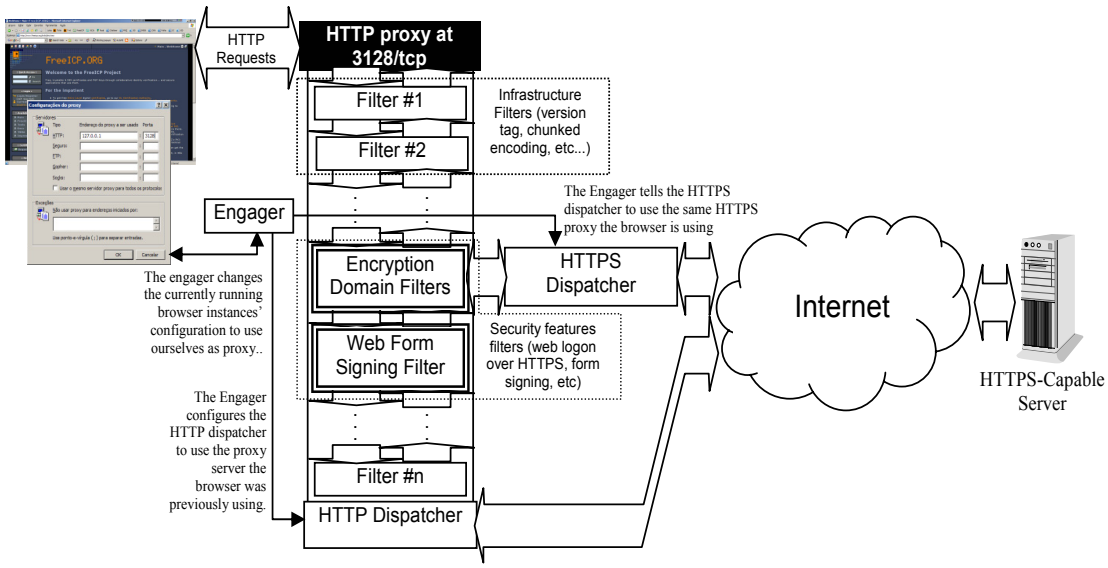



**Figure 1:** Overall architecture of the client proxy, which runs in the same computer as the browser. The engager changes the browser's proxy settings so that it uses our own local proxy. Before doing that, though, it detects which HTTP and HTTPS proxies the browser was using and configures our dispatchers to use them. This effectively puts us in the middle of the proxy chain. New HTTP requests originated by the browser will pass through our proxy, where our filters may act upon them. In fact, we have two filter chains, one for the outgoing requests and other for the incoming responses; some of them act upon the headers, other upon the bodies. Some features actually require several filters in cooperation to implement. If none of the filters actually consume the request (i.e., take it out of the chain), it reaches the default dispatcher in the end of the chain and it is sent as an HTTP request. The Encryption Domain filterset is a special set of filters that reroutes the requests that match certain criteria to be sent over as HTTPS. The HTTPS dispatcher makes use of the certificate store services (not shown in this picture) to validate the certificates and perform client authentication (with a friendly UI) if the site so requests.

## 2 OVERALL ARCHITECTURE

Our tool, code-named Kapanga, is an executable program that typically (although not necessarily) runs in the same computer as the user's web browser. A schematic depiction of its overall architecture can be seen in Figure 1. A brief description of its major components follows:

- **Certificate Store Manager (CSM):** provides all the underlying cryptographic services needed by all the other components. It manages and provides access to all the cryptographic objects (certificates, certificate revocation lists, private keys, signatures, etc) stored in various kinds of storage media (the local disk, removable storage devices, crypto-capable devices such as smart cards, etc); provides access to cryptographic algorithms and protocols. The CSM is detailed in section 2.1 .
- **Filtering HTTP Proxy Server**: receives the requests from the browser and feeds them through the filter chain. If no filters consume the request, it is passed to the HTTP dispatcher nearly unchanged. Filters may alter either the request before they're sent to the dispatcher or the replies berfore they're sent back to the browser. These changes implement the program's main features, as it will be detailed further along.
- **Engagers:** they are in charge of changing the HTTP proxy settings of all supported browsers to point to our own proxy described above, so that we get to intercept all HTTP traffic initiated by the browsers. Engagers are described in detail in section 2.3 .
- **Default Dispatcher:** an embedded HTTP user agent that sits at the end of the filter chain. It acts like a "default route" in a routing table: any requests that reach it are sent their destinations, either directly or via another next-hop proxy. It also proxies any authorization requests (e.g., Basic, Digest or NTLM authentication) that the next-hop proxy may require, so the authentication protocol is handled by the browser itself and any username+password dialog boxes that may be required is also shown by the browser itself. Upon receiving the results, it pipes them back to the response filters, which also play crucial security roles.
- **HTTPS dispatcher and the Encryption Domain:** similar to the default dispatcher, but tunneling the requests over TLS/SSL [2]. For performance reasons, it features support for rehandshakes and session caching [3]. It relies heavily on CSM services for validating the servers' certificates and providing client authentication if the server so requires. A request is sent through this dispatcher if the `host:port` of the request is listed in a set called *Encryption Domain* (this detour is actually accomplished by a special filterset collectively known as the

"Encryption Domain filters"). As with the default dispatcher, it may either send the request directly or use the `CONNECT` method to tunnel it over a next-hop proxy [5] if the engager has previously told it so. It is also responsible for sending back any authentication requests that the next-hop proxy may require.

*2.1 Certificate Store Manager*

Underlying the whole program is the Certificate Store Manager, providing crypto and PKI services to the other subsystems:

- **Certificate, CRL and private key enumeration and caching**: all those objects can live in one or more physical media. The local hard disk is called the primary store location, bringing a minimal set of certificates right from the installation procedure.

  The user may configure one or more secondary locations. Those are usually removable media, such as CD-ROMs, diskettes or USB flash memory devices ("pen drives"). Every three seconds or so the certificate store manager checks to see if these devices are readable and, if so, rescans them. This way, a user may have and use his/her certificates/keys in a removable storage medium during their entire lifetime[1].

  Crypto devices such as smartcards are also supported, although they are handled as special cases because some objects (private keys, primarily) may not be exported and we may only operate on them via the device's built-in cryptographic capabilities.

  The resulting in-memory cache can be seen as a concatenation of all the contents of all of the devices. CRLs are handled as a special case – since some of them tend to get very big, they are deallocated from memory as soon as the CSM is done using them in the trust calculations.

  Private keys are handled as special cases as well. When stored in crypto devices, the CSM directs all its crypto primitives to the device's drivers to make use of its embedded functionality; otherwise, they are loaded only when needed and the crypto primitives (signing/decryption) are directed to the software-based implementation.

  Our certificate store has another type of object called *attestation signature* or simply *attestation*. It is a signature block on someone else's certificate made by the private key of a user to indicate that it trusts that certificate (typically a root CA). This signature is detached – that is, it is stored in a separate file in a file format of our own devising; we will have more to say about attestations in section 2.1.1. .

- **Chaining:** after the certificates are loaded from the physical stores, the CSM tries to chain them. First, duplicates are discarded and certificates issued by the same CA are sorted by their `notBefore` fields and assembled as a doubly-linked list. The *best current* certificate is selected by applying two criteria: a) it is the one with the most recent `notBefore` and b) it must be still within validity (that is, with the current date/time before its `notAfter` field). If no certificate satisfies both requirements, we settle for the one that satisfies only (a).

  After that we build several indices for fast lookup: one keyed by the certificate's SHA1 hash, other by its Subject Key Identifier extension [7] and another by subject DN. This last one has a peculiarity: only the best current certificates make to this index; the future and previous editions don't get there.

  We then chain the certificates in the usual way, using the recently computed indexes to speed up finding the issuer of each certificate in the store (matching SKI/AKI pairs, when available, and by subject/issuer DNs as a last resort). We set the parent pointer of each certificate to the issuer and record its the depth in the tree (the whole chaining algorithm uses a breadth-first search precisely to make that trivial).

  CRLs are considered as an appendage to their issuer certificates and are chained to them. Private keys are also appendages and are linked to the certificates with the corresponding public key (the private key format stores the public key as well, so this comparison is straightforward).

- **Trust Status Calculations:** With all the certificates and associated objects properly chained, we start to verify their validitity periods, signatures of its issuers, attestations, etc. It is interesting to notice that all trust calculations are relative to the currently selected default ID, since attestations depend on it. Thus, whenever the user changes the default ID via the GUI, the whole trust statuses are recomputed. Section 2.1.2. describes each trust status in detail.

The CSM has a few other utilities and services:

- **Public CSM Server:** We coded a version of the CSM in a web server that is offered as an associated on-line service to the user and acts a public certificate/CRL repository. Over the years,

[1] Some of our users like to call this "the poor man's smartcard". We try to tell them this is a particularly nasty misnomer – not only because certain media such as USB "pen drives" are actually *more expensive* than smartcards (even including the cost of the reader), but also because they lack the tamper-proofness and crypto capabilities of the latter.

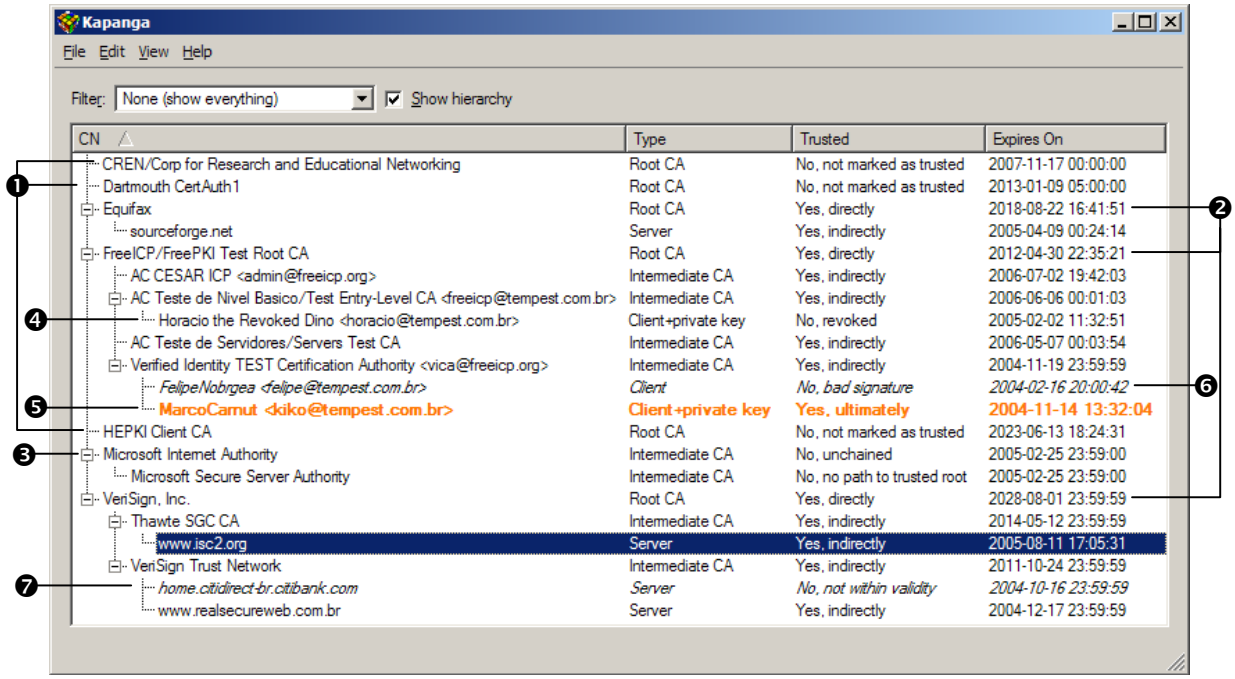



**Figure 2:** The CSM trust calculation results as displayed in the GUI. Here we have a certificate store with (1) three unattested roots and (2) three attested, trusted roots. There is also an Intermediate CA (3) that cannot be trusted because it's unchained, meaning that we lack its issuer root. All children of trustred roots are marked as indirectly trusted unless they've been revoked (4), their signatures don't match (6), or it's not within its validity period (7). The item marked in (5) is the current ID (aking to PGP's "default key"). All trust calculations are relative to the attestations (detached signatures on root CAs) this ID has previously performed.

we've been dumping on this server every CA certificate we lay our hands on. Whenever a certificate is unchained, the Kapanga may query the CSM server (either automatically due a configuration option or manually through a pop-up menu in the GUI) to see if it knows the missing issuer. The external CSM may also return CRLs – it has a built in robot that tries to fetch CRLs of all CAs it knows about from its distribution points.

- **Automatic CRL Download:** Just like the public online CSM server, the program's internal CSM has a similar feature – it automatically tries to download the latest CRLs from the addresses advertised in each CA certificate's `cRLDistributionPoints` extension. It can do so upon user request or automatically in the background. In the automatic mode, the list of candidates URLs is rebuilt each four hours (configurable) and we try to downloads CRLs from them. In case of success, the whole trust settings are recomputed and redisplayed. If some download fails, the next attempt time is subject to an exponential backoff algorithm with a maximum period of one week.

The overall effect we tried to achive is that the user doesn't have to worry about the intricacies of certificate management at all: he/she would only use the program features, collecting certificates along the way, and the CSM will do its best to ascertain its trust statuses and keep everything updated – without removing from the user the possibility of doing things manually if he/she so wishes.

### 2.1.1. Attestations

Attestations are signatures of a private key in someone else's certificates as a means of informing Kapanga that the owner of that private key trusts the signed certificate. They are akin to PGP's key signatures or introductions, except that they are stored in a file separate from the certificate itself.

We originally implemented attestations only for Root CAs as a more secure means to tell the CSM that particular CA is to be considered trusted. We were trying to avoid a simple vulnerability most browsers have: it's quite easy write a malicious executable that inserts a new fake root CA in their trusted certstore – in IE's case, it can be done in a few lines of code, since the root CAs are stored in the registry; for Mozilla-derived browser's, it requires only slightly more effort, since the root CAs are in a Berkeley-DB file.

As we will see in the next section, Kapanga trusts root CAs only if they're signed by the user's key. We say that the only truly trusted certificate is the user's own, because he/she has the corresponding private key. All the trust placed in the other certificates, even root certificates, stems from the user. Hopefully it also makes the root insertion attack slightly harder, for it will require the attacker to induce the user to sign the corresponding certificates.

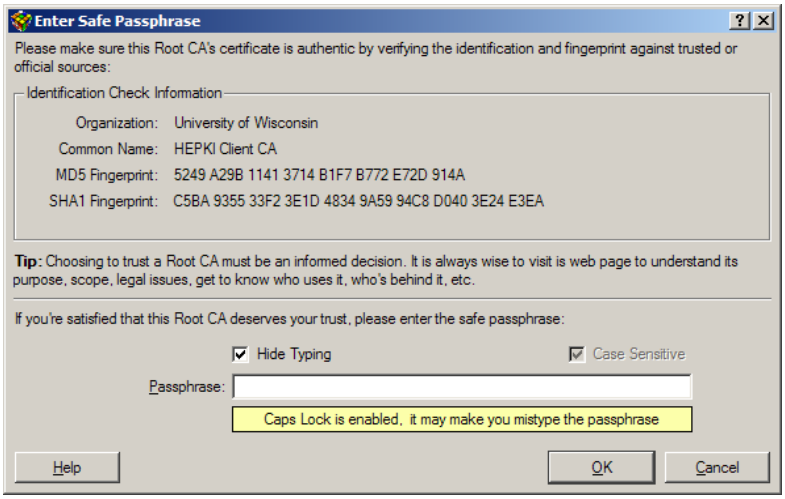



**Figure 3:** Manual Attestation Process. The user must sign the Root CA's certificate with his/her private key. Only then this CA will be considered directly trusted. The UI was designed as a single-step dialog box presenting the most important certificate identifiers for manual checking against other trusted sources, instead of the unecessarily complex and sometimes scary multi-step wizards most browsers have. The text succintly explains that this must be an informed decision. In this screenshot, we see the program requesting the private key's passphrase, which reinforces the sense of importance of this action. An always-enabled but impossible to disable check box reminds the user that passphrases are case-sensitive. Root CA attestations are also integrated with the certificate issuance/import dialogs, so users rarely come to this dialog to perform root attestations; it is more often used to attest certificates other than roots.

While this does improve security, this may seem as an extra complication: we *require* the user to have a certificate and private key; Kapanga is nearly useless if the user doesn't, since it will trust no one. After getting a certificate, the user would need to perform a few attestations. We tried to make this simple by integrating the attestation process with the certificate issuance/import processes: as shown in Figure 4, when the user gets a new certificate, a few checkboxes cause its root to be automatically attested, as well as all other roots this root trusts: root attestations on other roots are our bridge-CA mechanism.

Later on, we generalized the attestation system: the user now can sign any certificate he/she chooses.

This effectively makes Kapanga's trust system a cross-breed between the X.509's strictly hierarchical and PGP's web-of-trust models. While we were inspired by and tried to follow RFC 3280's certificate validation rules, we can't really say we follow them to the letter because it ended up evolving in a different direction.

Other interesting analogies can be drawn with other public-key based systems: for instance, signing an unchained server certificate in Kapanga is akin to adding a SSH server key to the `~/.ssh/known_hosts` file, except it's harder to spoof because of the signature.

### 2.1.2. Trust Statuses

The trust calculations assign one of the following statues for each certificate:

- **Ultimately Trusted:** this means that we have the private key corresponding to this certificate. Thus, it is an identity we can assume. Those certificates are considered to be "the root above the Roots", the true starting point all trust stems from. As a result, such certificates are considered trusted even if they're not properly chained or if its chain doesn't go all the way up to a trusted root; we say this status *overrides* the "Unchained" and "No path to trusted root" statuses described below.
- **Directly Trusted:** this means that this certificate has been *attested* by the current user. In other words, there is a signature block on this certificate correctly verified against the current user's public key as proof that the user gave his/her direct consent that this certificate must be considered trusted. If this a CA certificate, this causes all child certificates to be considered indirectly trusted.
- **Indirectly trusted: t**his means that the CSM has verified that the signature of the issuer is valid and that the issuer is trusted (either directly or indirectly).
- **Not Within Validity:** the certificate is not trusted because the current date and time is after the value specified in the certificate's `notAfter` field or before the value specified in the `notBefore` field. This status overrides all others (even the Ultimately Trusted stauts): the CSM doesn't even bother checking anything else.
- **Unchained:** the certificate cannot be considered as trusted because it we don't have its issuer. This status applies only to intermediate CAs and end-entities; it obviously can't happen in Root CAs. This status can override all the previous ones except the "Ultimately Trusted".
- **Not Attested:** this only happens to Root CAs. The certificate cannot be considered as trusted because we either have no valid attestation signature on this root from the current user's.
- **No path to trusted root:** the certificate cannot be considered as trusted either because the root of the chain has not been attested (it is not directly trusted) or some of its issuers are unchained (the chain doesn't go all the way up to a root CA).

- **Revoked:** the certificate is not trusted because its serial number is listed in its CA's Certificate Revocation List (CRL) and the CRL itself is valid.

The trust statuses for CRLs work a bit differently. A CRL is considered valid if the signature of its CA matches just fine, regardless of whether it is outdated or not. If a given certificate is listed in some CRL, it is flagged as revoked even if the CRL is not the freshest possible; the CRL checking engine tries to do the best with what it has. It is the responsibility of the automatic CRL dowload feature to try to keep CRLs as up-to-date as possible.

When the CRL checking engine is asked about whether a certificate is revoked or not, it returns an answer consisting of three items:

- **Is_revoked**: a tristate flag saying whether the certificate is revoked (true), or not (false) or if we can't ascertain because we have no CRL for this CA (unknown). If this flag is unknown, the remaining two items describe below are undefined.
- **Is_outdated:** it says whether the CRL used to compute the is_revoked status is outdated or not.
- **Reference date:** if is_revoked is true, it returns the revocation time and date as taken from the CRL. Otherwise, it returns the CRL's `lastUpdate` field, meaning that we can be certain that this certificate isn't revoked only up to the moment the CRL was issued.

#### 2.1.3. Certificate Issuance, Import and Export

Another important service provided by the CSM is providing support for having new certificates issued through a Certificate Authority. From the point of view of the CSM itself, its just a matter of having an RSA keypair generated and converting it to an Netscape SPKAC (Signed Public Key and Challenge, see [12]) format (a Certificate Signing Request would seem a better choice, but the reason for that will become clear further along).

From the point of view of the user interface, there are two very different implementations:

- The classic web-based style, in which the user directs his/her browser to the CA web page, fills some web forms and the browser activates the key generation procedure. Since this issuance system is intrisically intertwined with the filter system, it will be described along with our discussion of the HTTP filters in section 2.2 .
- We also wanted to have a PGP-like wizard-based instantaneous key generation. To that end, we implemented a specialized wizard that uses FreeICP.ORG's Entry-Level CA [4] to allow the user to get a free, instantaneous short-lived certificate. The rest of this subsection describes some particularities of this process.

In the first step, the user enters his name and email address, being also warned that the process requires being online or else the process will fail – this is unlike PGP. The user is also asked to reconfigure his/her spam filters to prevent the CA notification messages from being blocked.

After that, the wizard asks the CA whether the email address the user requested is already taken – that is, whether the CA has in its database a valid certificate issued for that email address. This is implemented by sending the CA a request for a "Revocation Reminder". If the server responds with a "No valid certificate associated with this email address" message, we let the user proceed. Otherwise we inform that the user is going to receive an email with revocation instructions and ask him/her to follow it before coming back to try to issue the certificate again. In this situation, the "Next" button of the wizard is grayed out, making impossible to proceed.

The next step is setting up the passphrase – historically the step users hate the most. This is constitutes a good opportunity to describe what kind of usability ideas we've been experimenting with, so we will detour from the "protocol nuts and bolts" approach we've been adopting so far and make an aside about our UI designs.

The philosophy is to try to steer the user to do the right thing, both through education and trying to prevent unwittingly dangerous actions. However, it can't be frustrating either, so the restrictions must not be absolute; they have to be bypassable, although the user must feel frowned upon when choosing the insecure path.

As usual, we have two passphrase text entry boxes. By defaut, they are set not to show the text being typed, replacing the characters by asterisks. Just like in PGP, however this is bypassable by unchecking a "Hide Typing" checkbox. This is needed because some poor typist users take too many attempts to make the content of the text boxes match that they become frustrated and quit. But unlike in PGP, if they opt to do this, they get a insistent blinking message warning them to make sure they aren't being watched or filmed.

We also implemented a warning about Caps Lock being enabled, now common in many programs.

Also common is the passphrase quality meter. The metering algorithm tries to estimate the entropy in the password roughly by making a weighted average of two criteria: the word entropy and the character entropy. The former is exceedingly simple-minded: we assume that each word adds about 11 bits of entropy. The latter is more complicated: we determine the bounding set of the characters of the passphrase in the

ASCII code space and use it as an entropy per character estimator. Then we multiply it by the number of characters and divide it by the efficiency of a customized run-length encoder. This has the effect of yielding very low scores to regular sequences such as "aaaa" and "12345". The quality meter displays its score in a progress bar and with a scale categorizing them as "simple", "good" and "complex".

The reason we didn't bother to be much more scientific than that with the quality meter is that in our early attempts it became clear it would result in it being overly frustrating to the end users. Our priority is to keep the users happy (or at least not too unhappy), so we calibrated (or rather downgraded) the algorithm many times to quell their complaints. We did perform some research about it, but in the limited time we had we could find no real good papers with general design guidelines for passphrase quality meters. We opted for trial and error based on the users' feedback.

In the end, we struck a middle ground with the following strategy: we made the meter slightly challenging and by default it doesn't allow you to go on if the score doesn't lie in the "good" range. However, we added a checkbox that allows you to disable the meter restriction altogether – in which case the user gets a polite message telling something like "now you're on your own risk, hope you know what you're doing – don't say I didn't warn you".

A frequent question our users pose is "what's a good passphrase anyway?" To try to answer that we implemented the passphrase suggestion dialog shown in Figure 4d. It generates passphrases suggestions using the Diceware method [18], which consists of generating a random number and mapping them onto a dictionary of 7776 words and common abbreviations, yielding 12.92 bits of entropy per word. (The method was originally designed to be performed by hand, pencil and paper using five dice tosses to select each word.) With 5 words we get more than 64 bits of entropy, which provides good enough protection against brute force attacks under quite general conditions while remaning reasonably easy to memorize.

Our user's feedback to the passphrase suggestion box has been a mixed bag. Some love it and many hate it – the primary complaint is that passphrases are way long. Many system administrators have been asking us to add a passphrase suggestion algorithm that matches their password policies like "8 characters with at least one punctuation character and a number not in the end nor the beginning". No amount of arguing that the diceware passphrases are more secure than those approaches seems to convince them. On the good side, however, our rejection rate has been zero precisely because we give the users the choice: they can simply disable the quality meter and ignore the suggestion box altogether if they really want to. Over time, we see that users gradually start to explore the passphrase suggestion box and the number of good passphrases slowly increases. A quantitative characterization of those intuitive perceptions may make fertile ground for a future paper.

The last page of the wizard is the one where the key pair is generated. As many other implementations do, a progress bar tracks the possibly lengthy key generation process; we were working on an educational animation to add to this window, but a discussion of its features is beyond of the scope of this paper.

After the key pair is generated, the private key is encrypted with the passphrase and saved in the Certificate Store. The public key is converted to the SPKAC format. When the wizard is invoked from the Keygen Interceptor filter (see section 2.2.2. ), we return this SPKAC to the filter. We also store along with the private key the state of the attestation checkboxes in the final page of the wizard (the CSM has facilities to add `property=value` tags along with any object) – they will be needed later when it's time to pick up the issued certificate and insert it in the CSM.

If, on the other hand, wizard has been invoked from the main menu, the SKPAC is sent in a HTTPS message to the FreeICP.ORG Entry-Level CA (the destination URL is configurable but with a hardcoded default). The Entry-Level CA responds immediately with the certificate in a PKCS#7 bag right in the HTTP response body.

### *2.2 Filters*

Filters are routines that change the request header, the request body, the response header or the response body of the HTTP requests received by our internal HTTP server. In our implementation, each filter can change only one of these items; the cooperation of several filters is often needed to implement a single particular feature. The filters are organized in two filter chains: the request chain and the response chain. Within a chain, the filters are executed sequentially in the order they've been set up. Some filters depend on others, so the chain setup tries to ensure that they are topologically sorted.

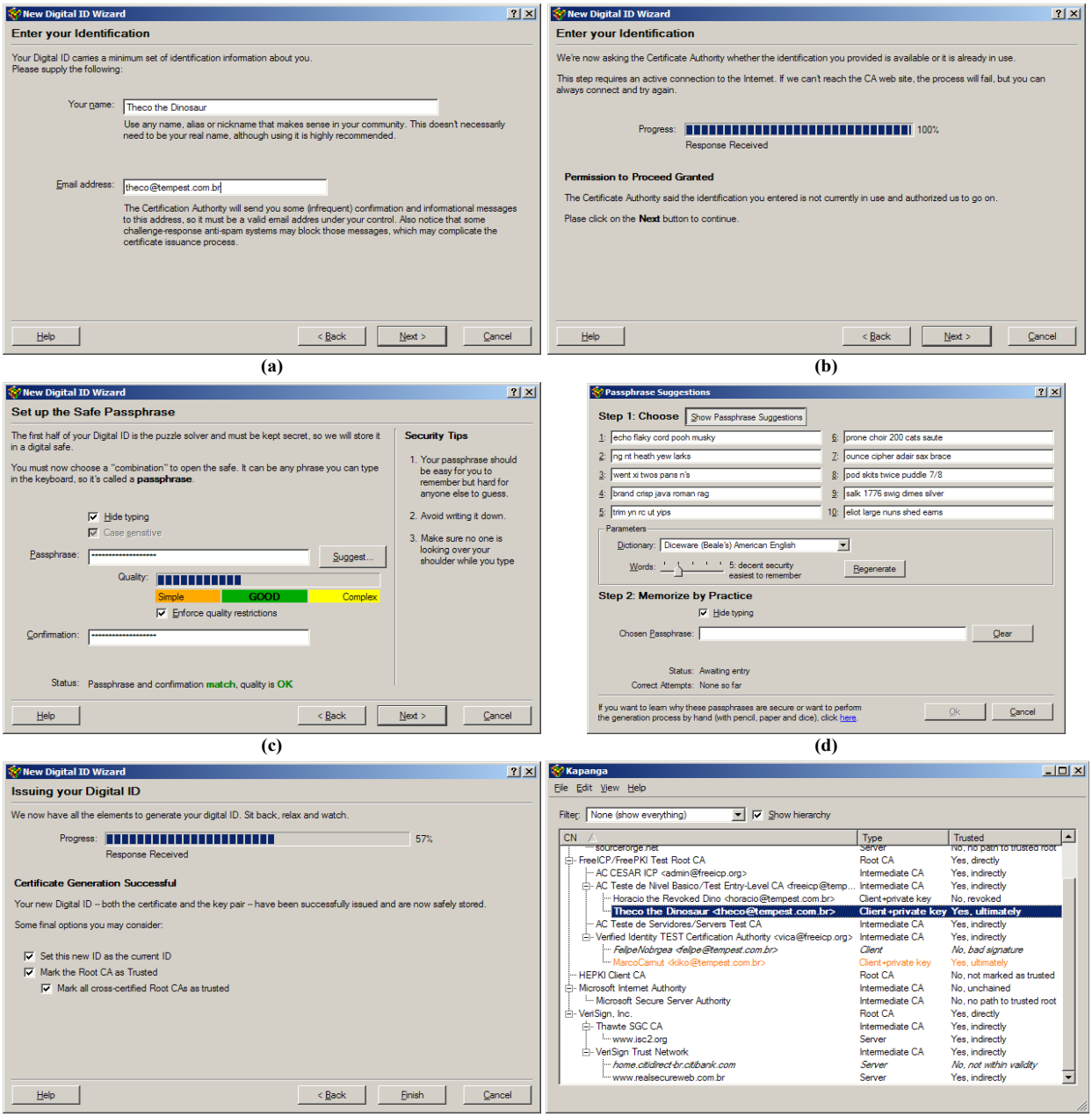



**Figure 4:** Wizard-style UI for using the FreeICP.ORG instantaneous Entry-Level certificate issuance process. In (a) the user enters his/her name and email address, while being advised the need to be online and that notifications will be sent over email. In (b) the CA is queried to see whether that email address is already in use. If so, the CA will send an email with revocation instructions and the process is halted. In (c) the user sets up a passphrase for the private key that is about to be generated. A quality meter gives prevents the user from choosing too weak a passphrase – unless the "Enforce quality restrictions" checkbox is disabled. The status texts indicate in real time when the confirmation matches and some educational security tips are also offered. In (d) we see the passphrase suggestions dialog: ten suggestions are put forth so that the user can choose visually without revealing the passphrase to shoulder surfers. As the first character is typed, all fields turn to asterisks. Each time the user correctly retypes the passphrase causes the chosen box to blink. Typing a different one resets the counter. Cheating by using copy-and-paste works but the user is politely warned that this doesn't help memorization. In (e), the key pair has been generated, converted to SPKAC, sent to the CA and the signed certificate has been received back. In (f) we see the new certificate and its associated private key in the certstore main window, already set as the default ID. The "Mark the Root CA as Trusted" checkbox caused the attestation of root certificate, so it's shown as directly trusted; the "Mark all cross-certified Root CAs as Trusted" checkbox caused an attestation on VeriSign's Root CA as well. The whole process takes 20 seconds or so for experienced users and less than two minutes for novices – most of it spent figuring out how to either please or bypass the quality meter. The user gets out of the process with all attestations already performed, so he/she will rarely have to perform manual attestations.

Notice that request filters can *consume* the HTTP request entirely, removing it from the chain so that it won't reach neither the subsequent filters nor the default dispatcher at the end of the chain. It then becomes this filter's responsibility to either issue the

request and insert the response back in the chain or to abort the request entirely.

Filters can be divided in two main groups described in the following subsections.

2.2.1. Infrastructure Filters

Infrastructure filters aren't directly involved in implementing the security features; they primarily provide services for the other filters. A description of the most important filters in this category follows, roughly in order from the simplest to the most complex:

- **Command Parser:** this is a simple request header filter that detects and extracts a special query string on the form "`x-kapanga-cmd=[command]`" from the URL. Below we have a short summary of the commands; each will be discussed in detail further along:

  `http://example.com/?x-kapanga-cmd=addsite(port,title,errpath)`
  Adds the current site (example.com:80) to the encryption domain. TLS/SSL connections will be sent to the TCP port specified in "port". If the certificate validation fails, the request is redirected to "errpath". The parameter "title" is a user-friendly added to the bookmark/favorites lists.

  `http://test.com:8080/?x-kapanga-cmd=delsite`
  Removes the current site (test.com:8080) from the encryption domain.

  `http://yasite.com/?x-kapanga-cmd=sign(data,sig)`
  Prepares to sign the field named "data" in an web form that will be downloaded as a result of this request. The signature will be performed when the user hits the submit button in his/her web browser and the result will be placed in a (possible new) form field named "sig" as a S/MIME signature.

  `http://somewhere.net/?x-kapanga-cmd=send-usable-ids`
  Forces the request to become a POST and sends a list of valid ultimately trusted certificates (without their respective private keys, of course).

  `http://whatever.org.ar/?x-kapanga-cmd=activate(sha1)`
  Sets the ultimately trusted certificate with fingerprint SHA1 as the default for client authentication with the server specified in the URL (in the example, "whatever.org.ar:80")

  `http://dummy.net/?x-kapanga-cmd=ua(string)`
  This command interacts with two filters. First, it tells the Version Tag filter to change the User-Agent header to *string*, effectively lying about the browser's identity and version. This will be needed to redirect us to the Nestcape-style certificate issuance system in commercial web-based CAs. Second, it arms the Keygen interceptor filter.

- **Version Tag:** A simple request header filter that appends an identifier and our version numer to the `User-Agent` header, without removing the browser's identification. This allows the web server to detect whether our tool is enabled and perhaps offer customized functionality. For instance, a client authentication-capable website could detect that Kapanga is engaged to the browser and offer its login URL already including the `x-kapanga-cmd=addsite` command.

  This filter is also responsible for "lying" about the browser's identity when the command parser has previously received the `x-kapanga-cmd=ua(string)` command. It changes all User-Agent request headers to the specified string (typically something like "Mozilla/5.0"). It also replaces all occurences of `navigator.appVersion` in JavaScripts by the specified string, since most web-based commercial CA software uses embedded scripts to determine the browser's version.

- **Encoding Dampers:** quells any encoding negotiation we can't understand, such as gzip or deflate encodings. In our current implementation, we don't support any encodings, so this is a simple filter that sets the the `Accept-Encoding` field of the HTTP request headers for the identity transformation. This is needed because several filters down the chain will need to parse the HTML when it comes back. This, of course, hinders any performance gains that those encodings might bring. Future implementations will replace the damper by a proper set of decoders.

- **Chunked Transfer Encoder:** converts the HTTP response bodies to the chunked transfer encoded form (see [1], section 3.6). This is needed because the response body filters will very likely change the length of the body, so the browser must not employ the ordinary strategy of relying on the `Content-Length` header. All that, in turn, is a consequence of the fact that the body filters perform on-the-fly rewriting, that is, they act upon each data block read from the network. The alternative would be to buffer the whole body, compute its new length after all filters had been applied and then send it along to the browser – a bad idea because response bodies can grow arbitrarily large, often several megabytes long, which would make latency too high and memory consumption prohibitive. The Chunked Transfer

Encoding scheme was invented precisely for this kind of situation when we don't know beforehand the size of the HTTP object we're transmitting.

An example may clarify what those filters accomplish. Suppose our browser issues the following HTTP request (indented for better readability):

```
GET http://testserver.example.com/t1.html?x-
kapanga-cmd=delsite HTTP/1.1
Accept: image/gif, image/x-xbitmap,
        image/jpeg, image/pjpeg,
        application/vnd.ms-excel,
        application/vnd.ms-powerpoint,
        application/msword,
        application/x-shockwave-flash, */*
Accept-Language: pt-br
User-Agent: Mozilla/4.0 (compatible; MSIE
            6.0; Windows NT 5.1)
Host: testserver.example.com
Connection: Keep-Alive
```

The full URL on the GET request gives away the fact that our browser was configured to use a proxy. This request also includes a Kapanga-specific command. After passing through the infrastructure filters, it would be sent over the network like this:

```
GET /t1.html HTTP/1.1
accept: image/gif, image/x-xbitmap,
        image/jpeg, image/pjpeg,
        application/vnd.ms-excel,
        application/vnd.ms-powerpoint,
        application/msword,
        application/x-shockwave-flash, */*
accept-language: pt-br
accept-encoding: identity;q=1, *;q=0
connection: keep-alive
host: testserver.example.com
proxy-connection: Keep-Alive
user-agent: Mozilla/4.0 (compatible; MSIE
            6.0; Windows NT 5.1) + Kapanga
            0.22
```

Since in this example Kapanga was not configured to relay the request to another proxy (that is, IE was not using a proxy before the engager did its job), the URL in the GET line is relative. Also notice that the command parser removed the "x-kapanga-cmd".

The encoding damper has also left its mark in the `Accept-Encoding` line telling that only the identity encoding is acceptable and all others are not. We can also see that the version tag filter added our name and version number to the `User-Agent` line.

After the request is issued over the network, the server responds with something like this:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 132

<HMTL>
 <HEAD>
  <TITLE>
   Infrastructure filters demo
  </TITLE>
 </HEAD>
 <BODY>
  <H1>Test!</H1>
  All's well.
 </BODY>
</HTML>
```

The chunked encoder converts this to:

```
HTTP/1.1 200 OK
content-type: text/html
transfer-encoding: chunked

40
<HMTL>
 <HEAD>
  <TITLE>
   Infrastructure filters demo
  </TIT
40
LE>
 </HEAD>
 <BODY>
  <H1>Test!</H1>
  All's well.
 </BODY>
</
5
HTML>

0
```

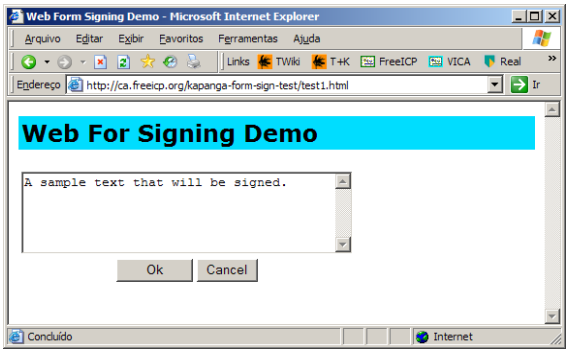
In this example, we lowered the maximum chunk size to 64 bytes to accentuate the encoding result; in our actual implementation, the maximum chunk size is 32Kb and it almost never gets that big because the networking layer sends it to us in even smaller chunks due to the underlying TCP buffers.

The chunk encoder filter has some heuristics to detect old browsers (such as IE3) that don't support the chunked transfer encoding. In those cases, it refrains from altering the body but it also quells the HTTP keepalive feature, so that the browser will rely on the connection termination to know when the body data finishes.

### 2.2.2. Feature Filters

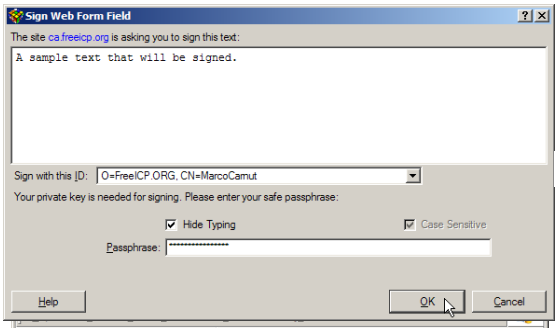
These are the filters that actually implement the security-relevant features, relying in the infrastructure provided by the previous filters and the CSM:

- **Web Form Signer:** this is a request body filter that acts only on POST requests with the "application/x-www-form-urlencoded" MIME type. It is activated when the command parser previously received a command of the form `sign(in,out,flags)`. The argument "in" is the name of a form field in the current page form which the filter will get data for signing. The filter displays a dialog box confirming the data being signed and requesting the passphrase for the private key that will be used for signing. When it receives these data from the user, it creates a S/MIME signed message and encodes as a (possibly new) form field named "out" (if "out" is ommited, it is assumed to be the same as "in"). The flags control things like whether we want our own certificate included in the signature, whether

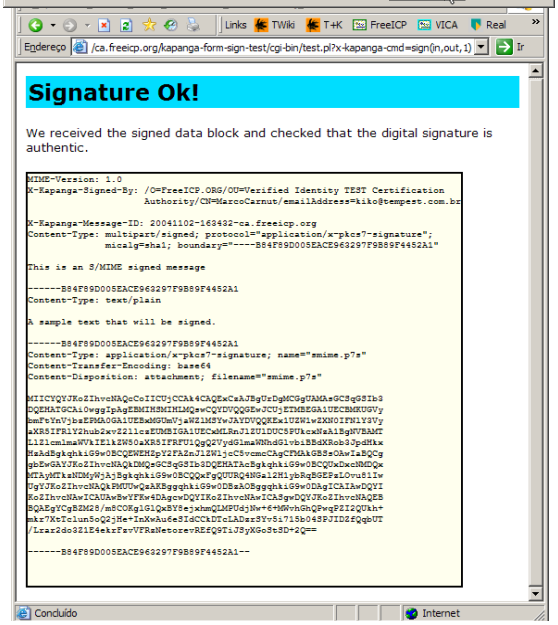



(a)

```
<HTML>
 <HEAD>
  <TITLE>Web Form Signing Demo</TITLE>
 </HEAD>
 <BODY>
  <table width=100% bgcolor=#00DDFF>
   <tr>
    <td>
      <font face="Verdana" size=+2><b>
         Web For Signing Demo
      </b></font>
    </td>
   </tr>
  </table>
  <form method=post
  action="test.pl?x-kapanga-cmd=sign(in,out,1)">
  <table border=0><tr><td>
<textarea rows=5 cols=40 name="in">
A sample text that will be signed.
</textarea>
</td></tr><tr><td align=center>
  <input type=submit name="submit" value="  Ok  ">
  <input type=reset  name="cancel" value="Cancel">
</td></tr></table>
  </form>
 </BODY>
</HTML>
```




(b)




(c)

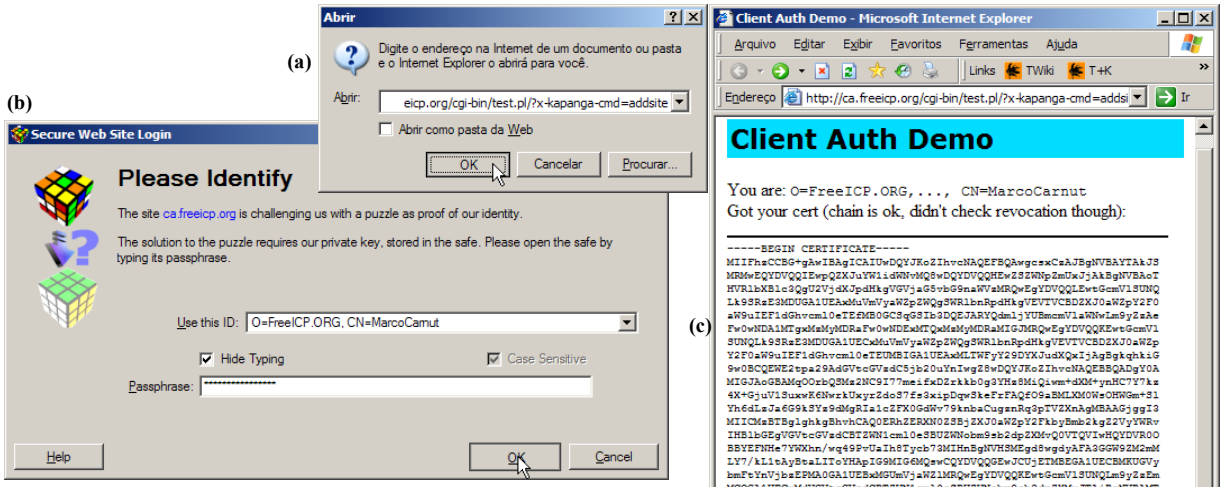
**Figure 5:** The web form signing filter in action. In (a) we see a minimalistic web in the browser and its source HTML. Notice the action URL with a Kapanga special command. The command parser field intercepts this command and set things up to intercept the POST request body and sign the "in" field, putting the result in a new form field named "out". The final one in the sign command is a flag to hasve the S/MIME signer not include the signer's certificate, just to keep the signature block small enough to fit this screenshot. In (b), the exact intercepted data that will be sign is shown in a dialog box, where the program allows the user to specify which key he/she wants to sign with and asks for the key's passphrase. In (c), the signature has been performed and sent to the server. A script in there displays the signed block for us. For sake of brevity, we have shown only the successfull case. Lots of failure conditions are handled as well – for instance, when the signature doesn't match, or the signed data has been changed by the client, when the user cancels without signing or when the proxy isn't activated.

to add the whole certificate chain up to the root, etc.

The advantage of this approach is that we can add form signing functionality to some web application just by activating Kapanga and making just minor changes in the web application – if it doesn't bother to verify the signature, it's just a matter of chaning the HTML to include the `sign` command and storing the "out" field somewhere. A signature verification engine, however, would be recommended to deal with exceptions such as invalid signatures or to ensure that the signed contents is the same as previously sent (since it's within the client's control, a malicious user may change it).

- **Usable ID enumeration:** This filter is triggered by the "send-usable-ids" command. First, it forces the request to become a POST (even if the browser has sent it as a GET or HEAD). Kapanga then builds a body with a list of PEM-encoded ultimately trusted certificates it has. This is extremely useful because the site can know in advance which identities we can assume, inform the user which ones are acceptable or not and help the user select an appropriate one for login or registration, reducing the likelihood of frustrating failures.

  The webmasters we have been working with point this particular feature as the one that mostly contributes for the overall user acceptance – it makes it viable to make helpful web-based certificate enrollment/registration system almost as simple as traditional name+password+cookie methods, as shown in Figure 7.




(f)

```
<HTML>
 <HEAD>
  <TITLE>
   Client Auth Demo
  </TITLE>
 </HEAD>
 <BODY>
  <TABLE width=100% bgcolor=#00DDFF>
   <TR>
    <TD>
     <FONT face="Verdana" size=+2><B>
        Client Auth Demo
     </B></FONT>
    </TD>
   </TR>
  </TABLE>
  <BR>
  You are: <TT>O=$SSL_CLIENT_S_DN_O,...,
              CN=$SSL_CLIENT_S_DN_CN</TT><BR>
  Got your cert (chain is ok, didn't check
  revocation though):<BR>
  <IMG SRC="https://ca.freeicp.org/vica/en/line.gif">
   <BR><font size=-1><TT>
$SSL_CLIENT_CERT
   </TT></font>
  <BR><IMG SRC="https://ca.freeicp.org/vica/en/line.gif">
 </BODY>
```

(d)

```
<HTML>
 <HEAD>
  <TITLE>
   Client A
  </TITLE>
 </HEAD>
 <BODY>
  <TABLE wi
   <TR>
    <TD>
     <FONT face="Verdana" size=+2><B>
        Client Auth Demo
     </B></FONT>
    </TD>
   </TR>
  </TABLE>
  <BR>
  You are: <TT>O=FreeICP.ORG,...,
              CN=MarcoCarnut</TT><BR>
  Got your cert (chain is ok, didn't check
  revocation though):<BR>
  <IMG SRC="http://ca.freeicp.org/vica/en/line.gif">
   <BR><font size=-1><TT>
-----BEGIN CERTIFICATE-----<BR>
MIIFhzCCBG+gAwIBAgICAIUwDQYJKoZIhvcNAQEFBQAwgcsxCzAJBgNVBAYTAkJS<BR>
MRMwEQYDVQQIEwpQZXJuYW1idWNvMQ8wDQYDVQQHEwZSZWNpZmUxJjAkBgNVBAoT<BR>
HVRlbXBlc3QgU2VjdXJpdHkgVGVjaG5vbG9naWVzMRQwEgYDVQQLEwtGcmVlSUNQ<BR>
Lk9SRzE3MDUGA1UEAxMuVmVyaWZpZWQgSWRlbnRpdHkgVEVTVCBDZXJ0aWZpY2F0<BR>
aW9uIEF1dGhvcml0eTEfMB0GCSqGSIb3DQEJARYQdmljYUBmcmVlawNwLm9yZzAe<BR>
```

**Figure 6:** The HTTPS Logon filterset in a client authentication scenario. In (a) the user directs the web browser to an HTTP URL containing the command for adding the site to the Encryption Domain. As Kapanga was engaged to the browser, the request is actually sent over HTTPS because the command parser filter is executed early in the filter chain. Thus, when the request reaches the HTTPS Logon filter, the site address and port is already in the Encryption Domain. In (b), the site has requested client authentication and Kapanga asks the user which certificate he/she wants to use and the passphrase of its associated private key. Unlike Internet Explorer, Kapanga doesn't show expired, revoked or altogether untrusted certificate, nor has a "remember password" checbox to ruin the whole security of the process. In (c), and the server have sucessfully completed the TLS handshake, sent the request and got the response back, where we see that the server sucessfully received and validated the user's certificate. In (d) we see the returned page source HTML; comparing with the source HTML template in (f), we can see that the absolute URL in (e) was rewritten (notice the change from "https" to "http") so that the image download would pass through our proxy as well.

Granted, this kind of enumeration may be abused by rogue sites to collect email addresses or tracking the user's habits. We argue this is a necessary evil to provide a seamless HTTP ⇔ HTTPS transition. Just in case, we left a configuration option that allows the user to either disable this feature entirely or get a popup then the site sends the enumeration command.

- **Remote ID activation:** this filter is trigged by the "activate(sha1)" command. It sets the preferred default ID for this site (as identified by the host portion of the URL) as the certificate with the specified SHA1 fingerprint. If we have no such certificate or if it's not ultimately trusted, no action is performed.

  This command is typically used in pre-logon page just before the "addsite" command to have the correct ID selected by default in the Web Site Login Dialog (where the user is prompted for the passphrase).

- **HTTPS Logon:** this filter is activated by the "addsite" command previously seen by the Command Parser filter. Recall that this command has three parameters: the SSL port (443 by default), a user-friendly site title/name and the error redirect URL.

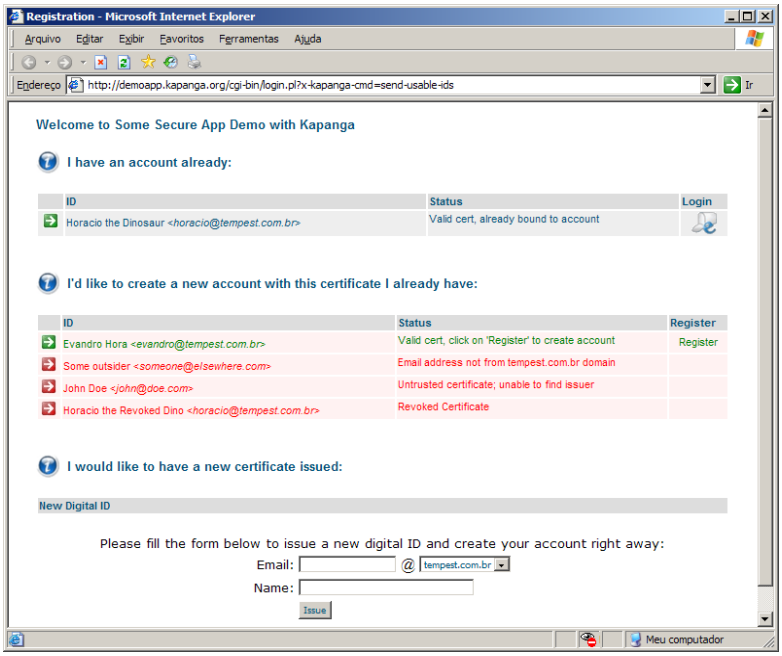



**Figure 7:** The `send-usable-ids` command allows the web application to provide friendly account creation assistance, explaining beforehand which certificates are acceptable and which are not and thus minimizing frustrating failures. The "login" and "register" links, use the `activate` command to force that particular certificate to be selected, minimizing user errors and then redirects to another URL with the `addsite` command, inserting the site into the Encryption Domain and starting the transition to the HTTPS site. Even so, the SSL handshake might still fail if the site's certificate doesn't pass Kapanga's validation. In this case (not shown in the picture) the "errpath" parameter in the `addsite` command would redirect the user back to a page explaining what went wrong and offering further help. At the bottom of the page, a form allows the user to start the wizard-based certificate issuance process directly from the web page: then clicking on Issue, the wizard pops up with the name and email fields already filled in.

The first thing this filter does is a purely user-friendliness action: it inserts the site URL and title in the Bookmarks/Favorites list (accessible via a menu), unless there is already a bookmark for this site there. That way, the user can easily come back to this site without having to remember the URL.

Then the filter inserts the site's address in the Encryption Domain, which is just a simple set mapping host:port pairs to SSL ports and error URLs. Since the Encryption Domain filter is right next in the chain, the request will be immediately rerouted to the HTTPS dispatcher.

- **Encryption Domain Filter:** this filter checks whether the host:port in the URL of the current request is in the Encryption Domain. If it isn't, the filter simply lets it follow its way on the filter chain, so it will ultimately reach the standard dispatcher and sent to the network over HTTP.

  Otherwise, the request is taken out of the chain (so it won't reach the standard dispatcher anymore) and fed to the HTTPS dispatcher, which, in its turn, starts the SSL handshake to the port specified in the site's entry in the Encryption Domain.

  If the server requests client authentication, the HTTPS dispatcher asks for the user's private key for the default ID set for this site; if this key is not cached, the CSM will display a dialog box stating the exact site name and prompting for the user's private key. If, on the other hand, the server doesn't require client authentication, this step is skipped.

  At this point in the SSL handshake, the HTTPS dispatcher receives the server's certificate. If the CSM deems the it untrusted or if any network or handshake error happened, the dispatcher displays a dialog box explaining the failure and returns down the response chain a redirect to the error URL specified in the site's entry in the Encryption Domain (if none was specified, the connection is simply broken). This way, the site has an opportunity to display a nice message telling the user that the HTTP ⇔ HTTPS transition failed and maybe provide options to retry or choose other authentication options (such as plain old name+password). This in direct contrast with popular web browsers, which simply break the connection on SSL failures, leaving the non-technical user wondering what went wrong.

  Finally, if no errors occurred and the certificate is held as trusted, the original HTTP request is sent over the HTTP tunnel and the response inserted back in the response filter chain. Also notice that, due to the SSL session caching, the whole verification process above happens only in the first connection to the site or when the cache entry expires.

- **URL Rewriter:** This filterset is actually part of the HTTP Logon filterset described above. It acts only on `text/html` MIME types on requests through the HTTPS user agent. Its main purpose is to rewrite URLs of the form:

  ```
  https://example.com/
  ```

  as

  ```
  http://example.com/
  ```

  Supposing, of course, that "example.com" is in the encryption domain. If the site name in the above

URL is not in the encryption domain, it is left unchanged.

That way, any further requests initiated by consequences of the browser parsing the current HTML will again be sent to us – recall that the engager configured us as the browser's HTTP proxy only – we do not receive any HTTPS requests the browser generates. So this filter tries to ensure that all URLs in the HTML the browser receives point to URLs with the HTTP scheme what we can proxy.

The discussion omitted several details for the sake of clarity. In our implementation, it is comprised of two filters: a response header filter for rewriting URLs in the Location field during redirect messages; and a response body filter the that parses the HTML looking for tags with `src`, `href` and `action` parameters and rewrites only URLs within them – that way, any URLs within the readable text (outside the tags) won't be touched. We also made it rewrite URLs in `window.open` JavaScript instructions, since it occurs quite often in many websites.

The response body filter is the system's Achiles heel: since it is static, it misses any absolute URLs generated dynamically by embedded languages such as Java, JavaScript or VBScript, nor it sees absolute URLs embedded in Flash movies or other plugin-specific objects. However, things work remarkably well in sites where nearly all embedded URLs are relative.

- **Keygen Tag Mangler:** This filter replaces the `<KEYGEN...>` tag used by Netscape-derived browsers in web forms to generate a new keypair [12] by a combo box (a `<SELECT>...</SELECT>` sequence in HTML parlance) allowing the user to choose one of the allowed key sizes. The original name of the KEYGEN tag is prepended with "x-kapanga-keygen-", so that the Keygen Interceptor field described below can intercept it. This filter is only active when previously told so by the command parser.
- **Keygen Interceptor:** this filter acts on response bodies of POST requests and only when the mime-type is "application/x-www-form-urlencoded". It looks for form fields with the name starting with "x-kapanga-keygen-". Upon finding it, it starts the New Digital ID Wizard right in the point where the user chooses the passphrase (see Figure 4c). When the wizard is done generating the keypair, it is converted to an SPKAC and sent over the form field with its original name (i.e., the "x-kapanga-keygen-" previously prepended is removed).
- **Certificate Interceptor:** this filter grabs the response bodies in "application/x-x509-{user,ca,email}-cert" MIME types. It also looks for these content-types in each section of multipart MIME types as well – this is the mechanism used by web sites and commercial web-based CAs to install certificates and certificate chains. The data is decoded (DER/PEM-armoured detection is built in and both single certificates and PKCS#7 bags are supported) and inserted directly to the Certificate Store.

  If one of the inserted certificates matches a previously sent SPKAC, the automatic attestations are performed. If the inserted chain contains a Root CA but no automatic attestation has occurred, a dialog box pops up informing the user that he/she may be interested in performing a manual attestation.

*2.3 Engagers*

The engagers are responsible for setting up the data interception in each browser by inserting ourselves in the proxy chain through the following process:

- The browser's current proxy settings are detected and saved for later restoration;
- The address and port of the proxy the browser is currently using for sending HTTP requests is detected and the engager signals our default dispatcher to use this proxy. If the browser isn't using any HTTP proxy, we tell our default dispatcher to do the same and send the requests directly;
- The browser's HTTP proxy settings are overwritten with "localhost:ourport", where "ourport" is the port where we've previously started a server to listen to this specific browser's requests;
- The address and port of the proxy the browser is currently using for sending HTTPS requests is detected and the engager tells the HTTPS dispatcher to use this proxy. Unlike the HTTP proxy, however, we don't overwrite the browser's setting.

Implementing this seems simple, but each browser presented its own special cases.

The engager for Internet Explorer proved to be the simplest to implement because IE has simple API calls to change the settings and have its currently running instances instantaneously reload any changes made to it. Slight complications arise due to the several versions of IE and the API itself. The most severe is with IE versions 5 and above: since it supports per-dialup connection profiles, each with its own proxy settings, the process above has to be performed for each dialup profile. In the end, the IE engager we implemented works with all IE versions all the way

back to version 3. Version 2 and below didn't support proxies at all.

Implementing the Mozilla engager, on the other hand, proved to be quite a challenge because of the lack of a simple way (as far as we know) to signal its currently running instances of any changes in its settings. Mozilla's settings are read once during program startup and kept in memory. We can easily overwrite the configuration files and its format is quite simple (although figuring out where it is located means messing with `registry.dat` /`appreg` file [10]). This works well for inactive profiles, but not for the active ones – the running instance doesn't notice that Kapanga (an external process, from its point of view) changed the files and thus doesn't reload them. And it overwrites our changes when saving the settings back to disk as it finishes.

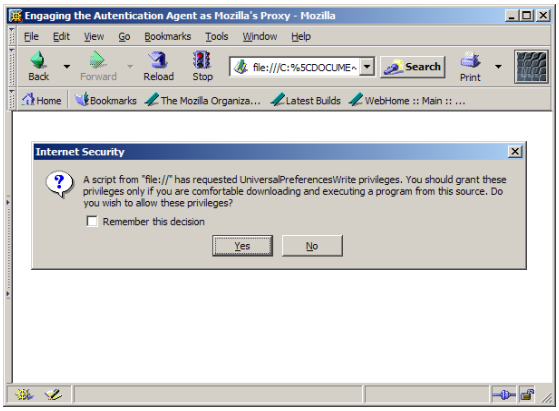



**Figure 8:** A JavaScript program is the only way to set the proxy settings in the currently running instances of Mozilla. However, since changing user settings is a privileged operation, its execution prompts a confirmation dialog box, somewhat thwarting the convenience of the engager.

The method we came up with works but is less than elegant: we generate a web page in a temporary, randomly-named file the local filesystem with a small JavaScript program that performs the changes. Then we direct the currenly running instances of Mozilla (via DDE [9] on Windows or X-Remote [8] protocol on Unix) to open this page. Since changing those settings requires granting special privileges to the script, the first time it is run Mozilla displays a confirmation dialog box, as shown in Figure 8 above.

The paranoid may regard this procedure as opening up a vulnerability itself – from there on, any local scripts (using the "file:///" scheme) may change Mozilla's preferences for that profile. It could have been worse, though: we considered and rejected the idea of avoiding the creation of a temporary file by sending the Javascript program over HTTP – that would mean the user should allow script execution over the "http://" scheme, which would open it up to abusive scripts from anywhere on the Internet.

A limitation of our current engager implementations is that they cannot handle Proxy AutoConfiguration [11], which is quite popular. Since implementing this support would require a quite capable JavaScript interpreter, we have chosen to deal with it in future versions; we felt that for the purposes of proving the concept, it was not essential.

Notice that engagers are just a convenience feature for users. They're obviously not necessary for the rest of the proxy to work, so long as the user changes the browser and Kapanga's proxy settings manually. That way, this whole system works even with browsers our implementation doesn't have specific engagers for. All that is required is that the browser must have proxy support. We've successfully run Kapanga in conjunction with many other browsers such as Konqueror, Opera and even Links (a console-based browser), just for kicks.

## 3 OTHER DESIGN ALTERNATIVES

Before settling for the particular set of design criteria and features we described, we considered and rejected a few other alternatives. While the reasons for some of them are pretty obvious, other are quite subtle and perhaps debatable. In the next subsections we describe a few choices we had to make and the rationalie behind tem.

### *3.1 Traffic Interception Method*

Using the browsers' native proxy support was an obivous choice – web proxy technology was specifically design to intercept and forward HTTP traffic and it's widely deployed and matured. Not that we lacked choices:

- Internet Explorer has a feature called Browser Helper Objects [15] that could make interception a lot easier on that platform because we wouldn't have to deal with next-hop proxies and its particularities (PAC, multiple authentication methods, etc). However, we didn't want to confine Kapanga's applicability to Windows only; as previously mentioned, we wanted it to work with any browser on any platform;
- Implementing Kapanga as a SOCKS [20] proxy might also work, but it would involve guessing port numbers where HTTP traffic goes. Besides, not all proxies support SOCKS;
- Redirecting the socket API calls would not only require the same port number guesswork, but it would require a lot more system-dependent code and it wouldn't allow Kapanga and the browsers to run on different machines.

### *3.2 The Pure-Scheme vs. Cross-Scheme Dilemma*

Kapanga uses what we call a *cross-scheme* system: when a site is in the Encryption Domain, we effectively map portions of the HTTPS scheme's

address space into the HTTP's address space. That is, the browser has no notion on whether the request is going through HTTP or HTTPS – this state information is in Kapanga's Encryption Domain. This has consequences:

- Bookmarks made when a site is in the encryption domain will probably not work when the site is not in the encryption domain or when Kapanga is not running at all (unless the site designer was very careful to handle this);
- We had to create the URL Rewriter Filter to force absolute URLs embedded in the HTML back to us. As previsouly mentioned, though, this fails with dynamically generated URLs.

Earlier in the design process, we considered – and rejected – what we called a *pure-scheme* system: we would actually implement two proxies, one strictly HTTP to HTTP and the other strictly HTTPS to HTTPS. Given that the namespaces don't collide, there would be no need for a URL Rewriter filter nor would we have problems with bookmarks.

This sounds like a good idea if we think only in terms of the pure HTTP proxy; however, given that SSL was specifically designed to be resistant to interception and tampering, the pure HTTPS proxy would have to be, in fact, a generic HTTPS spoofer/man-in-the-middle attack.

From a purely cryptographic point of view, this is quite easy to implement: during the initial SSL handshake, we send the browser a fake server certificate generated on the fly. From the user interface point of view, on the other hand, this has a problem: it triggers the browser's SSL warning dialogs, since the fake certificate isn't signed by a CA chain the browser trusts. This is clearly unnaceptable, not only in light of our philosophy of non-intrusiveness and minimum hassle for the users, but also because SSL-derived user interface problems are exactly what Kapanga was originally intented to solve in the first place.

We could make the SSL spoof work silently if we inserted a new root certificate in each browser's certificate store, but that would bring disadvantages: first, it would again limit Kapanga to run in the same computer as the browser (a restriction we didn't want to have); second, the exact mechanism for inserting new roots varies from browser to browser: IE stores trusted root CAs in the Windows Registry, while Mozilla-derived browsers use a Berkeley-DB file. This would increase the amount of platform-specific code Kapanga would have – something we've been trying to minimize all along –, not to mention that the process would fail if Kapanga runs without the proper privileges to write to those certstores.

There are other arguments against the SSL spoofer and the pure-scheme idea:

- Performance would suffer, since we'd have three encryption/decryption rounds: the browser encrypts the data, Kapanga would decrypt it, modify it and reencrypt it again;
- It wouldn't work on browsers without native SSL support; in contrast, the cross-scheme approach allows Kapanga to work even if the browser doesn't support SSL;
- Writing and releasing the code of a portable silent auto-engaging SSL spoofer would be more like giving a powerful weapon to the blackhats than a powerful protection to the average user.

Yet another advantage of the cross-scheme appoach is that we give the user the choice of not using Kapanga at all if he/she feels like, so we neither mess nor risk to break the user's web banking systems and other critical applications they already have running.

## 4 CONCLUSIONS AND FUTURE WORK

We described the architecture of a solution for perfoming the cryptographic and user interface aspects of HTTPS channel establishment and web form signature outside of the web browser. The key idea is to implement the crypto services in a proxy that rewrites the HTML on the fly and converts it to HTTPS when appropriate, so we can bypass the browser's and protocol limitations while retaining compatibility. Thus, any browser with proxy support can be used – the user is not forced to adopt any particular web browser. Another advantage is that our approach does not depend on any proprietary architecture such as ActiveX or Java.

Our primary motivation was to play with newer user interface concepts to make client-side PKI easier to use. A few results stand out: in other to make sites with client authentication that user's didn't hate, we had little choice but to address a few protocol and user interface gaps:

- A web site should be able to enumerate the user's certificate so as to offer assistance in registration as preparation for the HTTP-to-HTTPS transition (the SSL handshake with its certificate validation process);
- There had to be a way to redirect the user to an URL with a nice explanation, continuation options or alternative authentication methods when the SSL handshake fails. It's just not acceptable to break the connection and leave the user with a cryptic error message;
- The certificate issuance process shouldn't be so fragile as to break because of lack of ActiveX upgrades, different browser versions or the phase of the moon. Nor it should induce the user to store the private key without some effort to set up a decent passphrase. The process must be simple, reliable and hassle-free. Having it instantaneous is

a plus – with so many online services with instantaneous registration processes, it is hard to justify the severe identity validation procedures of most CAs;

- There really should be a simple way to do such a simple thing as signing a web form.

The implementation of those features in an external proxy enabled us to bypass the browsers limitations while providing the illusion that those features were "augmented" to the browser in an non-intrusive way. It also required minimal or sometimes no change to the server side at all: nothing needs to be changed for client-based HTTPS authentication; a simple change in the action URL in HTML forms enables form field signature (bigger changes may be needed if the application needs to validate the signatures); and small changes to the HTML page where the Nestcape vs IE issuance process decision is made is enough to support Web-based commercial CAs.

The price of this "backwards" compatibility is paid in the considerable complexity of the architecture and the horrible contortions our tool has to go about to implement them. Some problems may not have a good solution at all, such as the static nature of the URL rewriter filter not being able to handle dynamically generated URLs; this limits the tool's applicability to "well behaved" sites only.

There are many worthwhile future improvements on sight. Making the proxy work for generic TCP connections as well as HTTP may help extend the use of client-side authentication for several other protocols such as SMTP, POP3, VNC, X and many others. Extending the program to act as a Mail Transfer Agent (since every MTA is kind of a proxy) holds the potential for allowing us to make the same for mail clients: having the digital signature generation and verification be performed out of the mail client. Going further in this idea, we could also add support for encryption and decryption to allow message confidentiality. We are also working on making the CSM support PGP and SSH keys as well in order to achieve Jon Callas' concept of *format agnosticism* [21], consonant with our philosophy of bridging the PKIs together.

## 5 ACKNOWLEDGEMENTS

Thanks go to our co-developer Tiago Assumpção and to Felipe Nóbrega for implementing both the CAs used for the instantaneous certificate issuance and the public CSM server. We'd also like to thank Justin Karneges for writing the first versions of the Q Cryptographic Archictecture [19] on which our implementation is based. We are also grateful to the anonymous reviewers for their invaluable criticisms and suggestions for this paper.

Horacio and Theco are comic book characters from Mauricio de Sousa Produções that are arguably part of Brazilian pop culture, used here for entirely non-profit illustration purposes.

## 6 REFERENCES

1. Tim Berners-Lee et al., *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, 1999, http://www.faqs.org/rfcs/rfc2616.html
2. Eric Rescorla, *RFC 2818: HTTP Over TLS*, 2000, http://www.faqs.org/rfcs/rfc2818.html
3. Eric Rescorla, *SSL and TLS: Designing and Building Secure Systems*, 2001, Addison-Wesley, ISBN 0201615983.
4. Marco Carnut, Cristiano Lincoln Mattos, Evandro C. Hora & Fábio Silva, *FreeICP.ORG: Free Trusted Certificates by Combining the X.509 Hierarchy and the PGP Web of Trust through a Collaborative Trust Scoring System*, 2003, Proceedings of the 2nd PKI Research Workshop, http://middleware.internet2.edu/pki03/presentations/02.pdf
5. Ari Luotonen, *Tunneling TCP based protocols through Web proxy servers*, 1998, http://www.web-cache.com/Writings/Internet-Drafts/draft-luotonen-web-proxy-tunneling-01.txt
6. Steve Lloyd et al, *Understanding Certificate Path Construction*, 2002, PKI Forum, http://www.pkiforum.org/pdfs/Understanding_Path_construction-DS2.pdf
7. Steve Lloyd, *AKID/SKID Implementation Guideline*, 2002, http://www.pkiforum.org/pdfs/AKID_SKID1-af3.pdf
8. Jamie Zawinski, *Remote Control of Unix Netscape*, 1994, http://wp.netscape.com/newsref/std/x-remote.html
9. MSDN Library, *About Dynamic Data Exchange*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/WinUI/WindowsUserInterface/DataExchange/DynamicDataExchange/AboutDynamicDataExchange.asp
10. Daniel Wang, *Mozilla Profile registry.dat File Format*, http://wangrepublic.org/daniel/mozilla/registry
11. Nestcape Inc., *Navigator Proxy Auto-Config File Format*, 1996, http://wp.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.html
12. Netscape Inc., *Netscape Extensions for User Key Generation*, http://wp.netscape.com/eng/security/comm4-keygen.html

13. Arnold G. Reinhold, *The Diceware Passphrase Home Page*, http://world.std.com/~reinhold/diceware.html
14. Sean Smith, *Effective PKI Requires Effective HCI*, ACM/CHI Workshop on Human-Computer Interaction and Security Systems, 2003, http://www.cs.dartmouth.edu/~sws/papers/hci.pdf
15. Dino Esposito, *Browser Helper Objects: The Browser the Way You Want It*, 1999, Microsoft Corp., http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebgen/html/bho.asp
16. RSA Data Security Inc, *RSA Keon WebPassport*, `http://www.rsasecurity.com/node.asp?id=1230`
17. Verisign Inc., *Go! Secure for Web Applications*, http://www.verisign.com/products-services/security-services/pki/pki-security/pki-solution/web-application/
18. John Marchesini, Sean. Smith & Meiyuan Zhao, *Keyjacking: the surprising insecurity of client-side SSL*, http://www.cs.dartmouth.edu/~sws/papers/kj04.pdf
19. Justing Karneges, *Q Cryptographic Architecture*, http://delta.affinix.com/qca/
20. M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas & L. Jones, SOCKS Protocol Version 5, http://www.faqs.org/rfcs/rfc1928.html
21. Jon Callas, *Improving Message Security With a Self-Assembling PKI*, 2003, Proceedings of the 2nd PKI Research Workshop, http://middleware.internet2.edu/pki03/presentations/03.pdf