USB KEY PROFILE MANAGER FOR MOZILLA

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Yun Zhou

December 2004

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Chris Pollett

_____

Dr. Melody Moh

_____

Dr. Mark Stamp

APPROVED FOR THE UNIVERSITY

_____

ABSTRACT

USB KEY PROFILE MANAGER FOR MOZILLA

By Yun Zhou

Mozilla's profile manager allows users to save their private information such as bookmarks, cache and email drafts and customize their preferences settings. In this project, we build an XPCOM (Cross Platform Component Object Model) component called the USB (Universal Serial Bus) Key Profile Manager as an extension to the existing profile manager so that users can carry their profiles around via removable USB keys and use those profiles on different computers. To achieve this, three major functions were implemented: USB profile loader, user authentication and profile encryption. In this report, we will start with some background information and related work followed by a detailed description of the project design and an encryption performance test. We will summarize the report with a conclusion and a brief discussion of future work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

In this project, an XPCOM (Cross Platform Component Object Model) component USB (Universal Serial Bus) key profile manager was designed and implemented to enhance and extend the current functionalities of the Mozilla profile manager. With the help of the USB profile manager, user profiles stored on USB keys will be truly location independent and password protected. (In this report, the new component will be referred to as the USB profie manager, as opposed to the existing profile manager, i.e. the Mozilla profile manager).

Mozilla is one of the biggest open source projects at the time of this writing, and its code base keeps growing as more functionalities are integrated. Mozilla implements a full-fledged Internet browser with strong security protection, high performance, and good usability. As the descendant of Netscape, Mozilla's next generation browser Firefox has the capability of fitting seamlessly with multiple platforms and has therefore received much attention from the media and Internet users. The Firefox 1.0 release provides a rich yet customizable set of features such as live bookmarks, built-in popup blocker, Google search toolbar and development tools.

To achieve location independence, the USB profile manager automatically detects a mounted USB key and searches for profiles according to certain pattern. If profiles are detected, they will be registered with the browser so that the user can use them as

regular profiles. When the browser is shut down, the loaded USB profiles are removed from the registry so that no footprint will be left.

To protect the content of the profiles, the USB profile manager authenticates the user before a protected profile is accessed. When the user shuts down the browser or and switches to another profile, the current profile is encrypted using the AES (Advanced Encryption Standard) algorithm.

This project was developed on Red Hat Linux 9.0 with Mozilla 1.7 RC1. However, it can be developed on any other platforms or with other versions of Mozilla without extra difficulty. The USB key profile manager was implemented as an XPCOM component for backward compatibility. No modification was made to the existing code base of Mozilla. After installation, the new component is automatically registered with the browser and starts functioning. It can also easily be uninstalled.

The following is the outline of the topics being presented in this report. In Section II, we will introduce some important background knowledge and related work as a preparation for the readers to understand the design details presented in Section III. The background information includes the existing Mozilla profile manager, the concept of XPCOM components and the key points for developing such components, Mozilla's Network Security Services (NSS) and Personal Security Manager (PSM) that

provide the security algorithms needed for user authentication and encryption, the AES algorithm and some well-known hash functions. Additionally, a study on the read and write speed of USB drives is provided. In Section III, we will discuss the detailed design for the event-driven structure as the base structure of the USB profile manager, the implementation of the three key features: profile loader and unloader, user authentication, profile encryption and decryption, followed by a presentation of the sequence of operations. We then present a set of performance and usability test in the purpose of investigating whether the overhead is at an acceptable level. The report ends with a conclusion and a discussion of future work.

II. BACKGROUND AND RELATED WORK

2.1 Mozilla's Profile Manager

2.1.1 Strengths and Weaknesses

The profile manager is one of the key components of Mozilla. Our USB key profile manager is meant to be an extension of the existing profile manager. Mozilla uses profiles to store users' personal settings such as bookmarks, cookies, preferences, news group subscriptions, browsing history, cache, and so forth. Profiles not only help users manage their personal information, customize the browser's behavior in response to different events, but also provide big performance gain. For example, the usage of cookies and cache largely reduce the number of message exchanges between the browser and the content servers. Users are allowed to create multiple profiles and save

them to user-specified locations, such as removable disks. The profile manager provides a XUL-based (XML User Interface Language) user interface for the users to manage their profiles. The content of the profile can be easily accessed through the browser's link field.

However, the current profile manager has the following weaknesses that need improving. First, although profiles can be stored on removable storage medium, they do not really achieve location independence. This is because Mozilla does not recognize profiles unless they are registered. Registration can only be done when a profile is created, meaning that one cannot easily take a profile created on one computer and use it on another without copying and pasting all the files in the profile folder.

Second, the files in a profile folder is saved without any encryption and the profile folder is given a software-generated folder name ended with ".slt". Therefore, if an unauthorized user does a simple file search for folders with the pattern ".slt", she can easily locate the profiles and examine the content. This can be a security flaw because not only can private information be leaked, but also a hacker or malicious code and change certain security settings such as disabling strong encryption ciphers, allowing no encryption at all, or disabling pop-up warnings when a suspicious events occurs.

2.1.2 nsProfile

Most of the Mozilla profile manager is contained in the *nsProfile* class which implements the *nsIProfileInternal* interface. Our USB profile manager interacts with the Mozilla profile manager through the *nsIProfileInternal* interface. *nsProfile* accepts calls from other components to manage profiles. It can create, rename, copy and delete profiles. It also handles internationalization as well as migration of profiles created by former versions of Mozilla or Netscape. *nsProfile* communicates with the registry *nsProfileAccess*.

2.1.3 nsProfileAccess

The *nsProfileAccess* class handles directory access such as read and write operations on the profile registry. When the browser is started up, nsProfileAccess will create a *ProfileStruct* object and fill it with the information from the registry file. This information includes the name, location, the last modified date, creation time, email address, etc. When changes to the profile need to be updated or a new profile needs to be created, the *UpdateRegistry()* function will be called to write those changes to the registry file. This class also provides a set of "setter" and "getter" functions that forms an interface with the upper-level *nsProfile* class.

2.2 XPCOM - Cross Platform Component Object Model

2.2.1 Modularity

As mentioned earlier, the USB key profile manager was implemented as an XPCOM component. XPCOM is Mozilla's component architecture and it provides a set of core libraries that facilitates software development cross different platforms, including all flavors of Windows, Linux, BSD, MacOS, Solaris, HP-UX, AIX, and OpenVMS. It helps to modularize the code, increase flexibility, scalability, maintainability and reusability of the code. The components can be implemented independently and installed separately according to the user's needs. The versioning support enables developers to modify their components without having to recompile the entire project.

2.2.2 Interface

Similar to its counterpart, Microsoft's COM model, XPCOM is also interface-based. Components are typically compiled into dynamic reusable libraries and they communicate with each other at runtime through the interfaces. An interface acts as a contractual agreement and it decides how other components can access the component that implements this interface.

All interfaces are derived from a base interface *nsISupports* which tackles two major issues of interface-based programming: component lifetime and interface

querying. The *AddRef* and *Release* methods are responsible for adding and subtracting reference count. *QueryInterface* is for deciding whether a component implements a certain interface. The following is an example from this project that shows how to use *QueryInterface:*

```
nsCOMPtr<nsIDOMXULDocument> xulDoc = do_QueryInterface(domDoc);
if(xulDoc) {
// domDoc does implement nsIDOMXULDocument, so keep going...
}
else {
// domDoc does not implement nsIDOMXULDocument, so handle differently...
}
```

Figure 1: Use *QueryInterface* to decide whether an object implements an interface.

To identify a particular component, some kind of identifier needs to be assigned. XPCOM provides three types of identifiers. Here we will describe one of them briefly. Contract ID, the most frequently used identifier in this project, is a human readable string. The following is the contract ID of the USB profile manager:

#define USBPROFILE_ContractID "@azhou/USBProfile"

And here is an example of using the contract ID to get the service of Mozilla profile manager. It shows how we access this component in the project.

```
nsresult rv;
nsCOMPtr<nsIProfileInternal> profileMgr(do_GetService
(NS_PROFILE_CONTRACTID, &rv));
NS_ENSURE_SUCCESS(rv,rv);
```

Figure 2: Access Mozilla profile manager.

2.2.3 XPCOM Glue

To speed up component development, Mozilla provides a glue library which contains a set of generic macros and utility classes. By using these tools, our code is much simplified.

The smart pointer class *nsCOMPtr* is probably the most frequently used tool. As we all know, reference counting is a tedious, tricky and error-prone. When not handled properly, it will result in memory leak. *nsCOMPtr* is designed to take care of bookkeeping reference count for the developers so that they can focus on component specific code. For instance,

```
void doSomething() {
  nsCOMPTr<nsISample> sample;
  GetSample(getter_AddRefs(sample));
  ProcessSample(sample);
} // sample is automatically released when it exits the function scope.
```

The generic macros provide the common behaviors needed for all XPCOM components. For example, instead of implementing the *nsISupports* for every component, you just need to call *NS_IMPL_ISUPPORTS1* to have the glue library implement *QueryInterface*, *AddRef*, and *Release* for you.

Other tools include *do_QueryInterface()*, *do_GetService()*, *do_CreateInstance()*,

do_GetInterface(), *nsMemory*, *nsDebug* and *nsISupport* support.

2.2.4 Component Management

Mozilla's manages components mainly through three core component management helpers: *nsIComponentManager*, *nsIComponentRegistrar*, and *nsIServiceManager*. *nsIComponentManager* is responsible for creating components, and get a particular implementation of an interface. *nsIComponentRegistrar* handles the registration and unregistration of a component. In addition, it can discover newly installed components and enumerate registered components. *nsIServiceManager* creates and provides access to singleton objects. For most of the time, component developers do not have to use these interfaces explicitly, because most of the tasks can be achieved through the generic tools defined in the XPCOM glue, such as *do_GetService()* and *do_CreateInstance()* as mentioned in the above section.

2.3 Mozilla's Personal Security Manager (PSM) and Network Security Services (NSS)

2.3.1 Overview

This project makes use of some of the existing security functions provided by Mozilla in the Personal Security Manager (PSM) and Network Security Services (NSS) modules. PSM supports end-to-end security between the client, i.e., the browser and Web servers. PSM supports SSL (Secure Socket Layer) v2, v3 and TLS (Transport

Layer Security). It provides a large variety of cipher suites for key exchange, digital signatures, bulk encryption, and data integrity. Its sophisticated user interface allows users to manage their passwords, cookies, certificates for mutual authentication and customize their security settings. For instance, users can disable SSL v2, weak ciphers with small key size, or approve certificates only from certain certificate authorities. The user interface makes it very easy to understand as long as the user has some basic knowledge about Internet security. PSM also supports most of the PKI (Public Key Infrastructure) specifications.

2.3.2 The Structure of PSM

In the project, the *nsIHash* interface of the PSM module is used for creating password digest and user authentication. PSM includes two shared libraries: *pki* and *ssl*. The *ssl* package links to NSS 3.2, handles all the SSL sockets, provides event handlers and appropriate warnings, defines and implements IDL interfaces for access to NSS libraries. The *ssl* package also allows applications to use the cryptographic components without having to include other modules of the browser. Besides functionalities, the *ssl* package also offers good performance which is fast enough for disk encryption.

The *pki* package implements the user interface using XUL and related XPCOM objects.

2.3.3 NSS API

NSS provides an open-source implementation of security libraries that can be easily reused. For example, the USB profile manager encrypts password-protected profiles with the implementation of the AES algorithm provided in NSS. Besides AES (Advanced Encryption Standard), NSS supports a complete set of cryptographic algorithms such as SSL (Secured Sockets Layer) v2 and v3, TLS (Transport Layer Security) v1, PKCS (Public Key Cryptography Standards) #1, #3, #5, #7, #8, #9, #10, #11, #12, S/MIME for encrypted MIME (Multipurpose Internet Mail Extensions) data, X.509 v3 certificates, OCSP (The Online Certificate Status Protocol), and a suite of advanced ciphers such as RSA, DSA, Triple DES, DES, Diffie- Hellman, RC2, RC4, SHA-1, MD2, MD5. NSS also provides tools to manage keys and security modules, and to debug and diagnose code.

2.4 Advanced Encryption Standard (AES)

2.4.1 The Algorithm

As mentioned earlier, AES (Advanced Encryption Standard) is used in our project to encrypt profile folders so that they will be tamper-proof. AES is an iterated block cipher with a block size of 128 bits. It is considered a much stronger encryption algorithm than DES (Data Encryption Standard). AES allows keys with variable-length. Typical key sizes are 128-bit, 192-bit, and 256-bit. Depending upon the key

length, AES runs through 10 to 14 rounds of encryption, each of which contains four operations and three layers: an S-box for byte-for-byte subsititutions contributes to the non-linear layer; ShiftRow for rearrangement of bytes belongs to the linear mixing layer; MixColumnand also belongs to the non-linear layer; and MixAddRoundKey is for the key addition layer. [C. Kaufman, R. Perlman and M. Speciner].

2.4.2 Operation Modes

For block ciphers such as AES, encryption and decryption are done in the unit of blocks. If the size of the plain text is larger than the block size, the algorithm requires that the text first be divided into a number of blocks before the iterations. If the text size is not a multiple of the block size, we have to pad the text so that it is divisible by the block size. When the plain text is divided into blocks, there are a number of ways to encrypt these blocks. This is also known as operation modes.

The two best known modes are Electronic Code Book (ECB) and Cipher Block Chaining (CBC). These are also the modes that Mozilla's implementation of AES supports. The ECB mode is the simplest mode for which each block is encrypted and decrypted separately, and after all the iterations are done, these blocks are assembled to formed the cipher text or plain text. The formula of the ECB mode is the following (Ci stands for the cipher text for the i^{th} block; Pi for the plain text for the i^{th} block; K for the key, i = 0, 1, 2, ..., n) :

Encryption	Decryption
C0=E(P0,K)	P0=D(C0,K)
C1=E(P1,K)	P1=D(C1,K)
C2=E(P2,K)	P2=D(C2,K)
...	...
Cn=E(Pn,K)	Pn=D(Cn,K)

Table 1: The formula of the ECB mode.
Source: [C. Kaufman, R Perlman and M. Speciner]

ECB is known to be vulnerable to several attacks, such as block rearrangement. The biggest weakness of ECB is that it is relatively easy for attackers to gain information from the cipher text because the same block of plain text always results in the same cipher text with the same key.

Because of the flaws of ECB, the USB profile manager uses the CBC mode. This mode requires an IV (Initialization Vector). Here is how CBC works: the first block is XOR'ed with the IV before being encrypted. The cipher block is then XOR'ed with the second block before encryption. This process is repeated until all the blocks are encrypted. During decryption, the procedures are reversed. The formula is shown in Table 2:

Encryption	Decryption
$C0 = E(IV \oplus P0, K)$	$P0 = IV \oplus D(C0, K)$
$C1 = E(C0 \oplus P1, K)$	$P1 = C0 \oplus D(C1, K)$
$C2 = E(C1 \oplus P2, K)$	$P2 = C1 \oplus D(C2, K)$
...	...
$Cn = E(C_{n-1} \oplus Pn, K)$	$Pn = C_{n-1} \oplus D(Cn, K)$

Table 2: The formula for the CBC mode.
Source: [V. Klíma and T. Rosa]

The biggest advantage of CBC is that the cipher texts are different even if the plain texts are the same, so it exposes much fewer hints to attackers than the ECB mode. Rearrangement attacks are still possible with CBC, but it is much harder than with ECB [C. Kaufman, R Perlman and M. Speciner].

2.5 Hash Algorithms

2.5.1 MD5 and SHA-1

One way to authenticate a user is to compare the message digest of a given password against a saved digest of the correct password. For our project, a strong hash function such as SHA-1 is sufficient for generating message digests because it is a one-way function, meaning that given a digest generated by SHA-1, it is impossible to decide the original text without using a brute-force attack. Currently, the most popular hash functions are MD5 and SHA-1. MD5 generates a 16-byte hash and SHA-1 20-byte hash.

Another important issue for hash functions that needs to be considered is to avoid collisions of the hash results. However, recently two different texts have been discovered to produce the same hashes using MD5. Therefore, SHA-1 is applied in this project.

2.5.2 Dictionary Attack and Salted Hash

A common attack on passwords is the dictionary attack. Since users tend to use English words or passwords with some pattern such as birthday or lucky numbers, an attacker can guess all possible passwords, pre-compute their hashes and store them in a "dictionary". When the attacker captures a hash (which is usually not difficult because many systems save hashes at known locations), he can compare the hash against his dictionary to see if any match exists. Only one round of computation is needed for all possible passwords and the dictionary can be used repeatedly.

A salted hash can be used to make dictionary attack much more time and resource consuming. In this approach, a password is first appended by some random bits called salt then hashed. This salt is stored with the password for future authentication. If the same attacker obtains a salted hash, he has to re-compute all the guessed passwords with the salt. And he has to repeat the calculation for every different salt.

Due to the advantages of using salted hashes, it was chosen for generating password digest in this project. Details on this will be discussed in a later section.

2.7 Performance Test of a USB Drive

2.7.1 Advantages and Disadvantages of USB Drives

USB drives have gained much popularity due to their larger storage space, higher data transfer rate, smaller physical size compared to traditional floppy disks. USB drives are also more and more affordable and the prices keep dropping. At the time of this writing, one can purchase a 512 MB 2.0 USB key at around $60. With the advent of the 2.0 USB interface, the transfer rate will be much higher. In addition to these advantages, USB drives are also favored due the fact that newly manufactured computers usually don't come with a floppy drive any more.

2.7.2 Several USB Drives, Their Storage Space and Transfer Rates

The two characteristics of USB drives that are of major concern in this project are storage space and transfer rate. Here is a list of USB drive specifications.

Brand	Storage Space	Transfer Rate
Kingston Traveler USB 2.0 Hi-Speed Flash Memory	1GB	480 mbps
Lexar Media USB JumpDrive 2.0 Pro	1GB	4.5/6.0 mbps
Sony Corporation Micro Vault USB Storage Media	256 MB	5.5 mbps
*PNY Attache 2.0 USB Flash Drive	128 MB	4.0/5.0 mbps
CD Cyclone Flash Key	128 MB	800 kbps
EasyDisk USB Drive	128 MB	710 kbps
Sonnet Technologies Piccolo USB Flash Drive	256 MB	350/700 kbps
Linksys Instant USB Disk	128 MB	800 kbps

Table 3: Several USB drive products, storage space, and transfer rates
Note: * indicates the USB Drive being used for this projects.
Source:
http://accessories.us.dell.com/sna/productlisting.aspx?c=us&l=en&cs=RC968571&category_id=5949&first=true

The USB drives with lower transfer rates are most likely 1.1 USB drives. However, for 2.0 USB drives, if the underlying operating system does not support 2.0 USB interface, the transfer rate will be that of a 1.1 USB interface. USB keys with larger disk space and higher transfer rate definitely costs more. For example, the Kingston Traveler USB 2.0 Hi-Speed Flash Memory is priced at $295.95 at the time of this writing, while the PNY Attache 2.0 USB Flash Drive only costs about $40.

2.7.3 Read and Write Performance Test

The following table shows the different tests that have been carried out with regard to the read and write performance of a PNY Attache 128 MB 2.0 USB Flash Drive. The rates listed were the average rates of five experiments.

	Write Rate (Average) Output File Size: 98 MB Buffer Size: 1024 B	**Read Rate (Average) Input File Size: 97.7 MB Buffer Size: 1024 B**
USB Drive	7.168 mbps	4.944 mbps
Local Drive	83.152 mbps	156.32 mbps
Local Drive/USB Drive	11 times faster	28.8 times faster

Table 4: Read/Write performance test results

Although USB drives are much faster than floppy drives, they are still significantly lower in performance than local hard disks. However, as we can see from Table 3, faster 2.0 USB drives are coming out to the market. Although they are still costly at the moment, they will certainly be affordable in the near future.

III. DESIGN AND IMPLEMENTATION

In this section, we will discuss mainly the design and implementation involved with the component specific logic. For the issues on implementing interfaces common to all XPCOM components, please refer to [Turner & Oeschger].

3.1. Event-Driven Design

There are two mechanisms for the USB profile manager to cooperate with the existing Mozilla's profile manager and change the existing behavior. One is to change the existing profile manager and have it call the functions in the new component. These mechanisms were not chosen for this project because the existing profile manager is a substantial and mature module of Mozilla. Changes may not be easily accepted and may cause backward compatibility issues. My work is still preliminary and experimental. Currently the component is developed entirely on the Linux platform and is platform dependent, which is not compatible with Mozilla's cross-platform model.

In order to make the USB profile manager a pure add-on component which does not have any influence to the existing behavior when not installed, the component was designed based on event-driven mechanism. The component subscribes for the events that it is interested in, for example "*profile-approve-change*". When such event is generated a initiating component, that component will call the *Observe* function of all the subscribers to "notify" them about this event. So when the USB profile manager receives "*profile-approve-change*", it can react accordingly. Event-driven mechanism is a good way to decouple different modules of an application and yet still make the communication between them possible. Developers who implement the components that generate events do not have to have a priori knowledge about the observers of these events. Many XPCOM components use this way to get created at startup.

Mozilla provide a service called *nsIObserverService* for managing and notifying observers about all kinds of events. This interface defines three basic functions: *addObserver, removeObserver* and *notifyObservers.* Let's take the event "*profile-approve-change*" as an example to examine how events are handled via *nsIObserverService.* When the user wants to switch from one profile to another, the profile manager generates the "*profile-approve-change*" event get an approval from the interested components in order to proceed. It then calls the nsIObserverService to notify the observers. nsIObserverService maintains a list of observers for each event. It calls the *Observe* function of each of the components subscribed for this event, as

shown in Figure 3. If any component vetoes the change, the switching will be aborted.

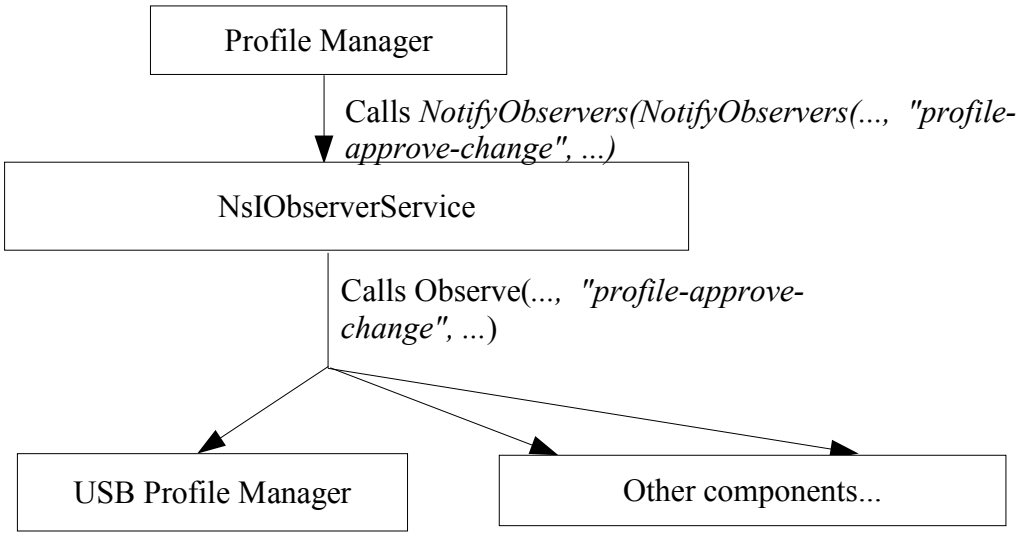


Figure 3: USB profile manager learns about profile switching

Any component that wants to receive notifications should inherit *nsIObserver* interface and implement the *Observe* function. If a component signed up for multiple events, it needs to check the topic of the event it just received in order to act accordingly. For example, the USB profile manager subscribe for three events: "*profile-approve-change", "profile-initial-state"* and "*xpcom-shutdown*". (The rationale of subscription for these events will be explained in a later section.) The following pieces of code show how to subscribe for events (Figure 4) and how to check the topic (Figure 5).

```
nsCOMPtr<nsIObserverService>  observerService(do_GetService
("@mozilla.org/observer-service;1"));
NS_ASSERTION(observerService, "could not get observer service");
if (observerService)
{
    observerService->AddObserver(this,
    NS_XPCOM_SHUTDOWN_OBSERVER_ID, PR_FALSE);
    observerService->AddObserver(this, "profile-approve-change", PR_FALSE);
    observerService->AddObserver(this, "profile-initial-state", PR_FALSE);
}
```

Figure 4: Subscribe for events.

```
if(strcmp(aTopic, "profile-approve-change") == 0)
{
    // Process...
}
else if(strcmp(aTopic, "profile-initial-state") == 0)
{
    // Process...
}
else if(strcmp(aTopic, NS_XPCOM_SHUTDOWN_OBSERVER_ID) == 0)
{
    // Process...
}
```

Figure 5: React according to the topic of a received event.

Every XPCOM component will be notified of the "*xpcom-startup*" be default, so does the USB profile manager. Therefore, the *Observe()* function acts like the *main()* function of an standalone application.

3.2 USB Profile Loader

The first major feature of the USB Profile Manager is to automatically detect and load profiles from mounted USB drives. As mentioned in the introductory section, Mozilla's existing profile manager does not recognize a profile unless it was created on the same local machine, because only in this way does Mozilla's registry have information about this particular profile such as the name and the location, etc.

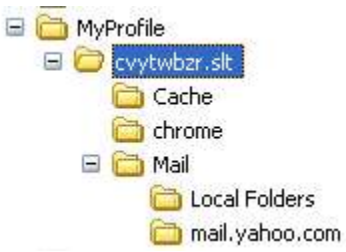
Therefore, the USB profile loader has to first detect mounted USB drives, then search for Mozilla profiles and at last add information to Mozilla's registry file.

3.2.1 Detect Mounted USB Drives

This part of code is the only part that is platform dependent. Making this feature platform independent is future work. We followed the same procedure as the UNIX command *df* employs. The idea is to examine the mount table and search for a mounted element with the device node called "*/dev/sda*". The *getmntent()* function returns important information about each mounted devices, including the device name and the path name. Therefore, the path name for the "*/dev/sda*" device is the path for a mounted USB drive. Note that we assume that only "*/dev/sda*" corresponds to a USB drive. However, this may not be true always. So our future tasks also include finding a more flexible way of detecting USB drives.

3.2.2 Search for Profiles on Mounted USB drives

Once a path name USB drive is located, a recursive function *SearchForProfile()* is used to search for Mozilla profile under this path. The way Mozilla arranges its profile folder is the following:



In the above example, the name of the profile is “MyProfile”. It contains the folder with a randomly generated name “cvytwbzr.slt”. All the profile files and subfolders are located in cvytwbzr.slt. Note that the name of every profile folder ends with “.slt”. Therefore, in this project, we determine whether a directory is a Mozilla profile by checking whether the directory name ends with “.slt”. A more sophisticated way is to check whether some important files exist and have the proper format. These files can be chrome.rdf, prefs.js, history.dat, etc.

3.2.3 Load Profiles to the Registry

Once a profile is located, we want to load the information into Mozilla's registry. Otherwise, the user will not see this profile in the profile selection dialog.

Mozilla's registry is a file based database that stores all the critical data about the browser, such as profiles, skins and locales. Its functionalities is similar to the Windows registry. This database maintains a tree structure, where each object is associated with a key. Possible operations on the registry include *addKey, getKey, removeKey, getString, setString* and so forth. The registry is a relatively low-level component of Mozilla. Ideally, add-on components should avoid manipulating the registry directly. It is preferred to add profiles through the existing profile manager. However, the *nsIPofile* interface does not provide public functions for registering a profile with an existing directory.

The data for the profiles forms a subtree of the registry. The string “Profile” is the name of the key for the base node of all the profiles. To get the node, the following is done:.

```
nsresult rv;
nsCOMPtr<nsIRegistry>          registry(do_CreateInstance
(NS_REGISTRY_CONTRACTID, &rv));
if (NS_FAILED(rv)) return rv;
nsCOMPtr<nsIFile> mNewRegFile;
rv = registry->Open(mNewRegFile);
if (NS_FAILED(rv)) return rv;
nsRegistryKey profilesTreeKey;
// Get the major subtree
rv = registry->GetKey(nsIRegistry::Common,
                      NS_LITERAL_STRING("Profiles").get(),
                      &profilesTreeKey);
```

Figure 6: Get the key for the base node of the profiles subtree.

To register a new profile, we first create a new key under the “Profiles” key.

```
// add the new node to the registry
nsRegistryKey profKey;
rv = registry->AddKey(profilesTreeKey, profileName, &profKey);
if (NS_FAILED(rv)) return rv;
```

Figure 7: Add a key for the new profile.

The new key is returned via *profKey*. We can then save the location of this profile through the *SetString()* function.

```
rv = registry->SetString(profKey,
                         NS_LITERAL_STRING("directory").get(),
                         profilePath);
```

Besides the location, there are other values to be set with the profile, such as the version number, the creation time, last-modified time and whether migration is needed. However, some of the information is hard to be extracted from the profile directory itself. So in my implementation, not all the information is provided when a profile is loaded into the registry.

3.2.4 Cleanup at Shutdown

Imagine a workstation which is worked on by many users everyday. If we keep adding USB profiles, soon the profile list will be very long and most of them will inaccessible without the correct the USB drive. So at Mozilla shutdown, all the USB profiles are removed from the registry. However, the user does not have to worry about losing the profiles, because the actual files remain untouched. And next time, when the same user starts Mozilla with the same USB drive, all the profiles contained in this drive will be loaded again transparently. On the one hand, removing the USB profiles will avoid cluttering the profile registry and reduce the performance overhead due to the large number of profiles the profile manager has to keep in the memory. On the other hand, this approach has the favorable side-effect that the user leaves minimal footprint on the local drive.

Removing profiles is much easier than loading them because it can be achieved by calling the *DeleteProfile()* function provided by Mozilla's profile manager. The following code shows how to remove all the USB profiles.

```
NS_IMETHODIMP nsUSBProfileManager::RemoveUSBProfiles()
{
    nsresult rv;
    nsCOMPtr<nsIProfileInternal>                profileMgr(do_GetService
(NS_PROFILE_CONTRACTID, &rv));
    if(NS_FAILED(rv)) return rv;

    PRUnichar **profileNames = nsnull;
    PRUint32 profileCnt = 0;
    rv = profileMgr->GetProfileListX(nsIProfileInternal::LIST_ALL, &profileCnt,
                                     &profileNames);
    if (NS_FAILED(rv)) return rv;

    for (PRUint32 i = 0; i < profileCnt; i++)
    {
            nsXPIDLString profilePath;
            rv = profileMgr->GetProfilePath(profileNames[i],
                                      getter_Copies(profilePath));
            if (NS_FAILED(rv)) break;

            // IsProfileOnUSB is a helper function for determining
            // whether a profile is on the USB drive
            if(IsProfileOnUSB(profilePath, PR_FALSE))
            {
                // PR_FALSE means leaving the profile files untouched.
                rv = profileMgr->DeleteProfile(profileNames[i], PR_FALSE);
                if (NS_FAILED(rv)) break;
            }
     }
    NS_FREE_XPCOM_ALLOCATED_POINTER_ARRAY(profileCnt, profileNames);
    return rv;
}
```

Figure 8: Remove USB profiles from the registry.

3.3 User Authentication

By implementing the USB profile loader, we have achieved location

independence. Mozilla users can now store their profiles on a USB disk and use it on another workstation where the USB profile manager is installed without the hassle of copying and pasting files over to a profile directory created on the local drive. However, it's known to all that USB keys are typically small and can easily get lost. Without user authentication, anyone who gets hold of the USB key with Mozilla profiles in it will have full access to all the private information, including certificates, mails, preference settings, bookmarks, history, etc. Therefore, we need to protect USB profiles from being misused or accessed without the owner's permission. But we still leave users to decide whether to protect their profiles or not.

3.3.1 Password Prompt

The password prompt serves two purposes: one is for the first-time signup, and the other is for login when a password has already be set up in an earlier session. Here is a snapshot of the dialog for each situation.

Figure 9.a: Prompt for the first-time signup.

Figure 9.b: Prompt for user login

When a USB profile is created, the user will be prompted for password protection. If the user clicks the “Cancel” button, the profile will still be created and started as usual. However, no additional protection will be provided, including bulk

encryption which we will discuss later in detail. Therefore any holder of the USB key can access this profile. It is obvious that unprotected USB profiles lead to much less computational overhead for encryption and decryption and the user does not have to remember a password. In Section IV, we will discuss how much the overhead can be.

To alarm the user of his choice, a warning message will be displayed.

Figure 10: A warning for an unprotected USB profile.

Each time when a user starts using an unprotected USB profile, the USB profile manager brings up the signup prompt and the user can protect the profile whenever he or she feels it is necessary.

If the user clicks the “OK” button without entering a password, the dialog will persist until a password is entered. Currently, there is no rule with regard to the set of allowed characters or the password length. And the user can apply the same password to different profiles, although this is not recommended because once the password is figured out by a hacker, all the profiles sharing the same password can be accessed.

When a user starts a protected USB profile, she will be prompted for login. If the user clicks the “Cancel” button, she will not be able to use the profile. If an invalid password is entered, an error message will be displayed and the profile is also not started.

Figure 11: Error message for invalid password.

3.3.2 How the Project Uses nsIWindowWatcher and nsIPromptService

The above dialogs are created via Mozilla's *nsIPromptService. nsIPromptService* implements a set of standardized prompt dialogs, such as dialog for confirmation, alert, password, both username and password, etc. For the USB profile manager, these dialogs are sufficient.

The prompt service is obtained through *nsIWindowWatcher.* Both *nsIWindowWatcher* and *nsIPromptService* belong to the Gecko embedding API [Gecko DOM Reference]. Gecko is the component in Mozilla that handles HTML parsing, page layout and the rendering of the application interface. It is a "fast, standards-compliant rendering engine that implements the W3C DOM (Document Object Model) [P. Le Hégaret, R. Whitmer and L. Wood] standards and the DOM-like (but not standardized) browser object model in the context of web pages and the application interface, or *chrome,* of the browser". We will discuss the DOM structure further in Chapter IV as we proceed to the technical details of the implementation. Gecko uses both native and XUL-based windows. XUL stands for **X**ML **U**ser interface **L**anguage. *nsIWindowWatcher* is responsible for creating and destroying windows, keeping track of the active window and sending notifications about changes to the windows. These operations are part of the embedding code and are transparent to the other components. Therefore, a component using a certain window only needs a weak reference to it. The following code shows how to use *nsIWindowWatcher* to get a reference to the prompt service and how to prompt for user login.

```
nsIWindowCreator *creatorCallback = new nsWindowCreator();
if (!creatorCallback)
        return PR_FALSE;

nsCOMPtr<nsIWindowCreator>  windowCreator(NS_STATIC_CAST
(nsIWindowCreator *,
        creatorCallback));
if (windowCreator)
{
    nsresult rv;
    nsCOMPtr<nsIWindowWatcher>                                  watcher
(do_GetService(NS_WINDOWWATCHER_CONTRACTID));
    if(NS_FAILED(rv)) return PR_FALSE;
    watcher->SetWindowCreator(windowCreator);
    nsCOMPtr<nsIDOMWindow> activeWindow;
    watcher->GetActiveWindow(getter_AddRefs(activeWindow));
    nsCOMPtr<nsIPrompt> prompter;
    watcher->GetNewPrompter(activeWindow, getter_AddRefs(prompter));
    prompter->PromptPassword(title.get(), instructionText.get(),
                             getter_Copies(password),
                             nsnull,
                             nsnull,
                             &retval
                             );
}
```

Figure 12: Get the prompt service via *nsIWindowWatcher* and prompt for the password

Note that when calling *GetActiveWindow(),* we pass the current window, the profile selection dialog in this case, as the parent window of the prompt dialog. Otherwise, system will crash when the prompt persists for a non-empty password. The password entered by the user is stored in the *password* argument. To check whether the user clicked “OK” or “Cancel”, test the *retval* argument.

3.3.2 Salted Password Digest

When the user enters a password for the first time, a SHA-1 hash is created for the password. Then a random integer, called “salt” is generated and appended to the hash and another SHA-1 hash computed and saved under the profile directory. The first hash is used as the key for the AES encryption, which will be explained in the next section. The second hash is used for password validation. The salt is also saved at the same location. The next time when the user tries to log in, after the first hash is generated, it is appended by the saved salt and hashed again. Then the second hash is compared against the saved digest. If they match, login succeeds; otherwise, an error message “Invalid user!” is displayed and the user is rejected to use the profile.

Considering the weakness of MD5 mentioned in Section 2.5.1, the SHA-1 algorithm is selected for computing the hash. A salt is used to make dictionary attack a lot more time-consuming. There are two ways of making use of Mozilla's hash algorithms. One is by calling the NSS library directly, the other is by using the functions available in the *nsIHash* interface of PSM's SSL packet.

The *nsIHash* interface provides four functions that are sufficient for calculating hashes. The *Create()* function creates a computation context for a particular hash algorithm that the user selects, SHA-1 in our case. The context contains an input

buffer as The SHA-1 context is defined in *sha_fast.h* of the NSS *freebl* library.

```
struct SHA1ContextStr {
union {
 PRUint32 w[80];          /* input buffer, plus 64 words */
 PRUint8  b[320];
} u;
PRUint32 H[5];           /* 5 state variables */
PRUint32 sizeHi,sizeLo;     /* 64-bit count of hashed bytes. */
};
```

Figure 13: The definition of the SHA-1 context.
Source: mozilla/security/nss/lib/freebl/sha_fast.h

The *Begin()* function does some preparation work for a new round of hashing. *Update()* updates the context with new information such as the input to be hashed and the hash length. The *End()* function finishes the computation. The result is stored in first argument. This entire procedure is shown in Figure 14.

```
PRInt32 size = (PRInt32)strlen(thePassword);

nsresult rv;
nsCOMPtr<nsIHash> mDataHash = 0;

 // Initialize the crypto library. SHA1 hash has length 20
PRInt16 mHashType = nsIHash::HASH_AlgSHA1;
mDataHash = do_CreateInstance(NS_HASH_CONTRACTID, &rv);
if (NS_FAILED(rv)) return rv;

rv = mDataHash->Create(mHashType);
if (NS_FAILED(rv)) return rv;

rv = mDataHash->Begin();

// Compute the hash...
int status;
if (mDataHash)
{
        mDataHash->Update(thePassword, size);
        status = PR_GetError();
        if (status < 0) return NS_ERROR_FAILURE;
}

PRUint32 hashlen;
mDataHash->ResultLen(mHashType, &hashlen);

(*computedHash) = (unsigned char *)PR_MALLOC(hashlen+1);
if (!(*computedHash)) return NS_ERROR_OUT_OF_MEMORY;


// Finish computing
mDataHash->End(*computedHash, &hashlen, hashlen);
status = PR_GetError();
if (status < 0) return NS_ERROR_FAILURE;
```

Figure 14: Compute the SHA-1 hash of a given password.

3.4 AES Encryption and Decryption

3.4.1 Encryption and Decryption

Once the user authentication feature is in place, we prevent random users from accidentally using other people's protected profile without knowing the password. However, this does not prevent people from peeking into the profile files through a simple text editor, since all the files are stored in plain text. Hence, an encryption algorithm must function together with user authentication for a profile's privacy to be fully protected.

3.4.1 AES Encryption Functions in the NSS Library

USB profile manager encrypts protected profile with the Rijndael AES algorithm implemented in the NSS *freebl* library because AES is much stronger secret key encryption algorithm than DES. This part of code gives an example of how to use the NSS library directly.

We only need to call three functions to have NSS encrypt a file for us. First, we need to call *AES_CreateContext()* to create the context. This function takes a key, an IV and the operation mode as part of the arguments. The possible AES encryption modes implemented in Mozilla are the ECB mode and the CBC mode. The CBC mode is applied by USB profile manager because it is a safer mode. (Please refer to chapter II for a more thorough discussion.) An AES context is defined in *rijndael.h* as shown

in Figure 15. It keeps information needed for the computation such as the IV, the key and the block size.

```
struct AESContextStr
{
  unsigned int  Nb;
  unsigned int  Nr;
  PRUint32      *expandedKey;
  AESFunc       *worker;
  unsigned char iv[RIJNDAEL_MAX_BLOCKSIZE];
};
```

Figure 15: The definition of the AES context.
Source: From Mozilla's *nss* package, mozilla/security/nss/lib/freebl/rijndael.h

Nb indicates the size of a block in bytes. *Nr* is the number of rounds each block encryption will go through. *worker* indicates whether to use encryption function or decryption function.

After creating the context, we call *AES_Encrypt()* and pass the context variable and in the text input to actually execute the algorithm. The output text and output length are stored in the argument variables. Note that the output character array must be allocated with enough space to hold the entire output. Finally, we destroy the context and free up the space by calling *AES_DestroyContext()*.

The decryption procedure is similar except that *AES_Decrypt()* is called.

3.4.2 Generate the Key and the IV (Initialization Vector)

AES allows variable key sizes. We only use 128-bit key length in the current version. The first SHA-1 hash generated from the password as mentioned in Section 3.3.2 is used as the key for the AES encryption. To obtain the IV, we compute the hash for the concatenated string from the password and the salt. The SHA-1 hash contains 20 bytes and only the first 16 bytes are used for encryption.

3.4.3 Pad the Input Text

When the input plain text is larger than the block size which is the same as the key size, the text is divided into multiple blocks and encryption is repeatedly applied to each of the blocks using the CBC mode discussed in Section 2.4.2. Mozilla's implementation of AES assumes that the size of the input text is divisible by the block size. But this is usually not the case. So it is the responsibility of the caller of this function to pad the input text so that its length is a multiple of the block size before calling *AES_Encrypt().* Otherwise, this function will abort. The USB profile manager pads the plain text with '\0' when necessary.

Padding should not be necessary because the size of the cipher text generated from *AES_Encrypt()* is always a multiple of the block size. But it is a useful practice to check the size in case that the cipher text has been modified.

3.4.4 Encode the Cipher Text in the Base64 Format

The cipher texts are encoded in the base64 format before written to the files. Before decryption, the based64 texts are decoded back to the binary format. We can take advantage of Mozilla's routines for base64 encoding/decoding defined in NSS *util* packet.

3.4.5 Example: A Plain Text and Its Encrypted Version

In this section, we will show a plain text and its encrypted version in the base64 format. See how long it takes you to find the key and the IV.

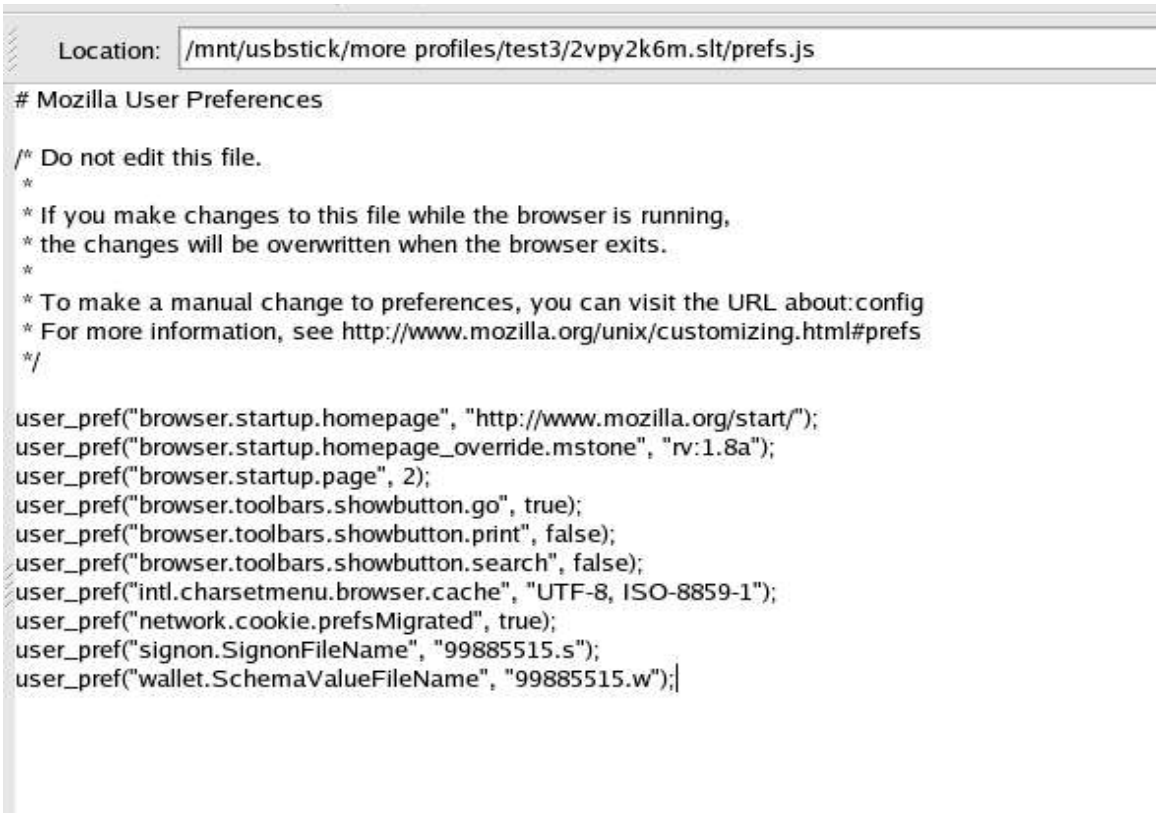


Figure 16.a: A prefs.js file.

Figure 16.b: The same prefs.js file after AES encryption and base64 encoding.

3.5 Detailed Sequence of Operations

In this section, we will discuss the sequence of user authentication, profile decryption and encryption in more detail and the rational behind this sequence which is not as straightforward as one would think.

3.5.1 User Authentication during Profile Switch

The procedure for successful user authentication is straightforward. But if the password is invalid, the user is rejected the access to the profile. There are two cases we need to consider separately: the user switches from one profile to a protected USB

profile; the user tries to access a protected USB profile at startup. Why do we have to handle these cases differently? It is because Mozilla's profile manager generates different set of events in these situations.

First, let's consider the case where the user tries to switch from one profile to a protected USB profile but provides an incorrect password. Our USB profile manager then need to block the access and the user should stay with the previous profile. The cleanest way is to prompt for the password if the switching actually take place. Therefore if the login fails, we can keep the old profile. As we mentioned earlier, all our operations are event-based. So let's first examine the available events we can make use of and then determine which event is the most appropriate one to subscribe for. Here is a list of events the Mozilla profile manager generates according to *nsProfile.cpp*:

"profile-approve-change"
"profile-change-net-teardown"
"profile-change-teardown"
"profile-before-change"
"profile-change-net-restore"
"profile-do-change"
"profile-after-change"
"profile-initial-state"

The actual switch happens after the *"profile-before-change"* event. If we veto the the change at the *"profile-approve-change"* event, no attempt is made to change the profile settings. This is the cleanest way to reverse the switch. What's more, Mozilla provides

the *nsIProfileChangeStatus* interface with a *VetoChange* function. So all the USB profile manager needs to do is the following:

```
nsCOMPtr<nsIProfileChangeStatus> status(do_QueryInterface(subject));
if (!status)
    return NS_ERROR_UNEXPECTED;
status->VetoChange();
```

Figure 17: Veto the change of the profile when login fails

However, there is one challenge due to this procedure. If the switch has not taken place, we cannot get the name of the target profile by simply calling *GetCurrentProfile()* provided by the Mozilla profile manager. The profile name is required for finding the corresponding password digest. The only way to get the name is via the user interface and find out the user selected profile name. This is where we need to understand the DOM structure employed by Gecko. Figure 18 shows the collaboration diagram for the n*sIDOMWindow* interface.

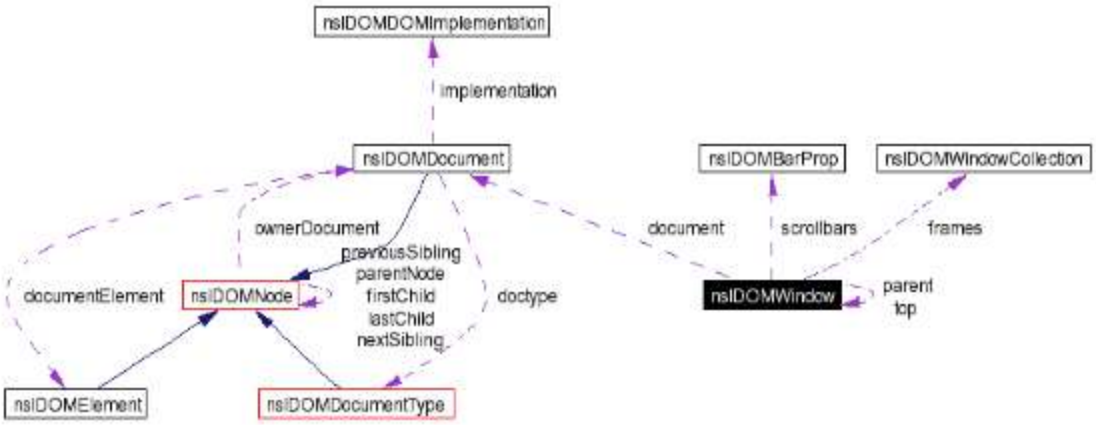


Figure 18: Collaboration diagram for the n*sIDOMWindow* interface
Source: "Mozilla Cross-Reference".
http://lxr.mozilla.org/mozilla/source/dom/public/idl/base/nsIDOMWindow.idl

The "Profile Selection" dialog is a DOM window. Every DOM window contains a document object which is an abstraction of an HTML, XML, SVG or XUL document. The "Profile Selection" dialog contains a XUL document object called *nsIDOMXULDocument*. Each DOM document has a group of *nsIDOMElement* objects which can be identified by IDs. To find the elements contained in the DOM document of the "Profile Selection" dialog, we need to look into the *profileSelection.xul* file. One will notice a listbox element with the id "profiles" which may well be the id of the profile list. According to the class hierarchy, *nsIDOMXULSelectControlElement* is a subclass of *nsIDOMXULElement* which in turn inherits *nsIDOMElement*. Once we get a reference to the profile list object through the "profiles" id, we can cast this *nsIDOMElement* object into an *nsIDOMXULSelectControlElement* object by the *do_QueryInterface()* macro function and call GetSelectedItem() of the

nsIDOMXULSelectControlElement object in order to get the selected profile item. The follow piece of code shows the entire procedure of getting the name of the user selected profile.

```
nsCOMPtr<nsIDOMWindow> activeWindow;

watcher->GetActiveWindow(getter_AddRefs(activeWindow));
NS_ENSURE_TRUE(activeWindow, NS_OK);

nsCOMPtr<nsIDOMDocument> domDoc;
activeWindow->GetDocument(getter_AddRefs(domDoc));
if (domDoc)
{
   // See if we contain a XUL document.
   nsCOMPtr<nsIDOMXULDocument> xulDoc = do_QueryInterface(domDoc);
   if(xulDoc)
   {
      nsCOMPtr<nsIDOMElement> domElement;
      nsString identifier = NS_LITERAL_STRING("profiles");
      rv = domDoc->GetElementById(identifier, getter_AddRefs(domElement));
      if(NS_FAILED(rv)) return rv;
      nsCOMPtr<nsIDOMXULSelectControlElement>
                        xulSelect(do_QueryInterface(domElement));

      nsIDOMXULSelectControlItemElement *selectedItem;
      rv = xulSelect->GetSelectedItem(&selectedItem);
      selectedItem->GetLabel(selectedProfile);
   }
```

Figure 19: Get the name of the user-selected profile via the DOM window object.

For a complete listing of the class hierarchy of the DOM structure, please refer to http://unstable.elemental.com/mozilla/build/latest/mozilla//dom/dox/inherits.html.

3.5.2 User Authentication during Startup

When user authentication fails at startup, we cannot veto the change as we do while switching to a new profile, this is because at startup, the Mozilla profile manager does not generate the “profile-approve-change” event. So the USB profile also needs to subscribe for the “profile-initial-state” event in this case.

The first approach attempted was to show the “Profile Selection” or “Profile Creation” dialog again when an invalid password is entered. However, this approach did not work out because the browser exited no matter what the user selected in the dialog. The cause for the termination of the browser was that when we failed the access to the profile, the first “Profile Selection”dialog shown by the Mozilla profile manager always returned false.

The second and the current approach is to let the USB profile manager prompt the user for a profile selection before the existing profile manager does. If authentication succeeds, the USB profile manager will set the flag *StartWithLastUsedProfile* to be true. When this flag is set, the Mozilla profile manager will not show the “Profile Selection” dialog, but start with the user-selected profile instead.

When authentication is unsuccessful, the USB profile manager will either show the “Profile Selection” dialog or the “Profile Creation” dialog if there is no other

profile is available. Note that the "profile-initial-state" event is generated after the Mozilla profile manager has already loaded the information of the target profile. Therefore, we need to tell the profile manager to shut down the profile via the *ShutDownCurrentProfile()* function.

3.5.3 Profile Decryption and Encryption

The profile directory is decrypted once the user passes the authentication test. However, there is still a caveat when the user accesses a secure USB profile at startup: Mozilla tries to locate the *prefs.js* file before decryption is done. Since the browser cannot find *prefs.js* in the directory of the target profile, it uses the default *prefs.js* settings. So after we decrypt the profile, we have to tell Mozilla to use the decrypted *prefs.js* file. This can be done by calling *ReadUserPrefs()* of the *nsIPrefService. ReadUserPrefs()* takes an *nsIFile* argument which represents the preferences file that we want the browser to use.

When the user switches from a protected profile to another profile, we need to encrypt the first profile. One would think that encryption of the previous profile can be done right after user authentication at the "profile-approve-change" event. But this is not a good approach because some of the profile files such as *bookmarks.html* are not updated when this event is sent out. In order to catch all the updates, the USB profile manager sets a *waitForEncryption* flag at the "profile-approve-change" event and

waits until it receives the “profile-initial-update” event to encrypt the previous profile. It also need to remember the key and the IV for the profile to be encrypted.

3.5.6 Flow Chart

Figure 20 illustrates the sequence of operations at startup.

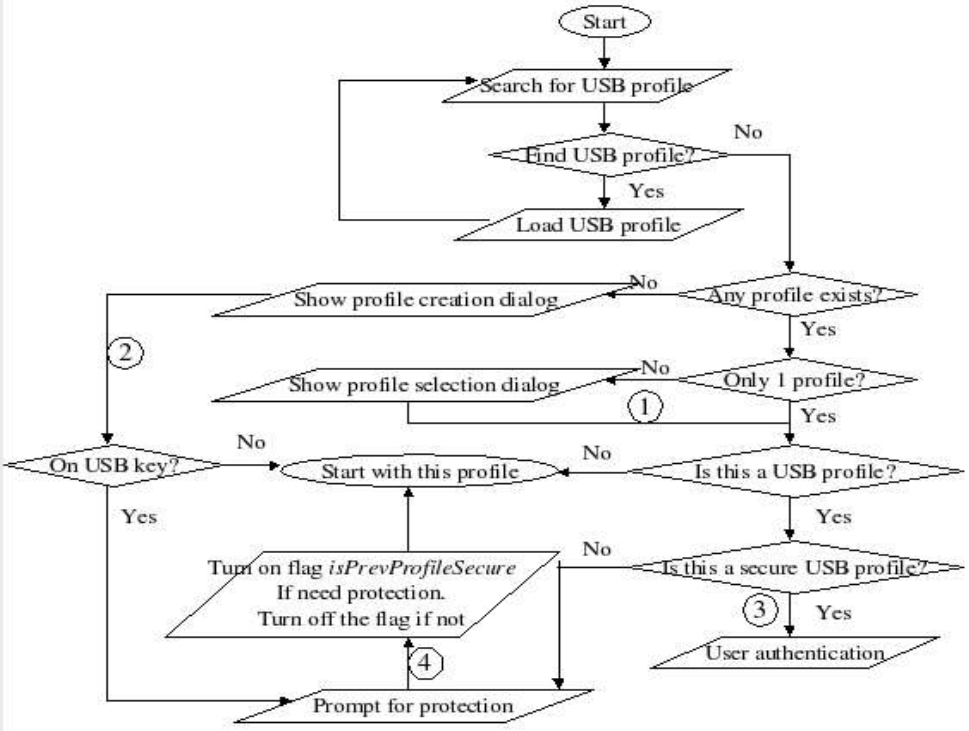


Figure 20: The sequence of operations at startup.

The operation labeled “3” in Figure 20 is achieved by calling the Mozilla profile manager's *SetCurrentProfile()* function which will generate the “profile-initial-state”

event. The USB profile manager has a *isStartup* flag set to be true by default. *isStartup* becomes false when the user successfully started up with any profile. When the *SetCurrentProfile()* function returns, if the *isStartup* flag is still on, we know that the user did not provide a valid password. Then the browser will exit.

The logic for handling different cases while opening a target profile is implemented in the *PrepareForSwitchingProfile()* function. As mentioned earlier, the *isStartup* flag is used to determine whether the user is accessing a profile at startup or during profile switch. We should distinguish between these two cases because they trigger different events.

The operation labeled "4" in Figure 20 sets the *isPrevProfileSecure* flag to true if the user has selected to protect this profile. Otherwise, this flag is turned off. This flag is used to determine whether we need to encrypt the profile when the user switches from it to another profile or shuts down the browser.

The following figure shows the zoomed-in sequence of operations of the "User authentication" step in Figure 20. Note that after the step "Show profile selection dialog", we follow the same sequence as indicated by step "1" in Figure 20; after the step " Show profile creation dialog", we follow the step "2" in Figure 20.

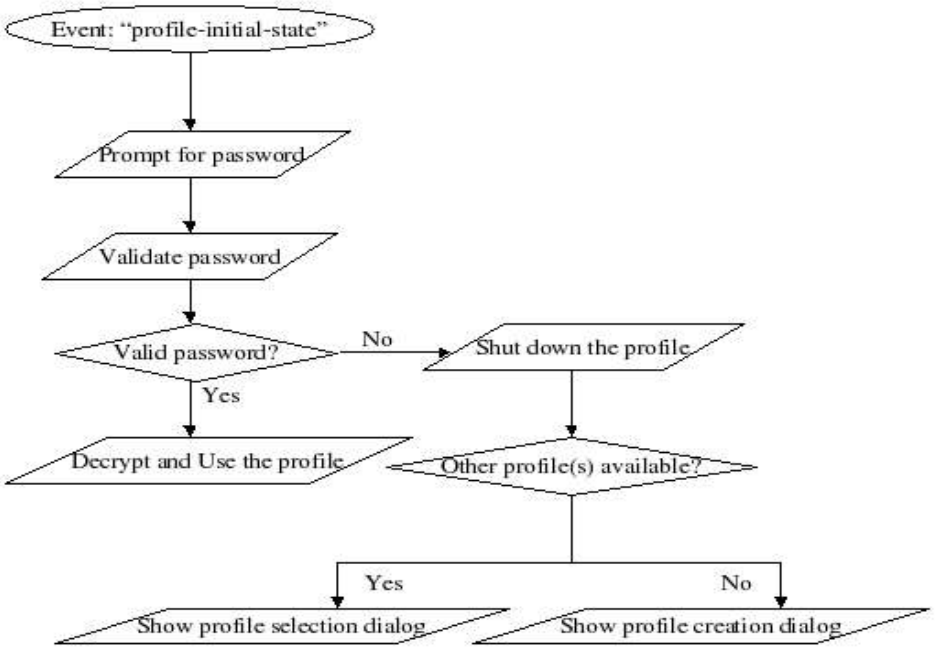


Figure 21: User authentication and profile decryption at startup.

When the user switch from one profile to another. There are several points worth discussion. First, whenever the USB profile manager decides to proceed with the new profile, it has to check whether the flag *isPrevProfileSecure* is on. If yes, it sets the *waitForEncryption* flag so that then it is notified of the "profile-initial-state" event, it will remember to encrypt the previous secure profile. If the profile that the user switches to is also a protected profile, we need to set the flag *isPrevProfileSecure* back

on. This flag is also checked during Mozilla shutdown.

Also note that the USB profile manager needs to remember the directory path, the key and the IV of the previous profile in order to do the encryption.

IV. PERFORMANCE AND USABILITY TEST

Security protections add overheads to the existing functionality. Those overheads include computational cost and I/O costs during encryption and decryption. When the profile is large, for example over 10MB, this overhead can become significant. This situation can happen when the user saves many email drafts, or the cache contains dozens of files. In addition, encryption and decryption are in-memory operations. Therefore, when encrypting large files, if the memory is not large enough, it can become a bottleneck and reduce performance dramatically.

The following is a list of profiles with different numbers of files and sizes and the time it takes to encrypt the entire folder. The test was carried out on my personal computer Mobile Intel Celeron CPU 2.40 GHz with 192 MB of RAM. As mentioned earlier, in order to reduce the overhead, the USB profile manager only encrypts files that have been modified during the session. This feature is turned off in this performance test. Also note that the elapsed time in the figure include the time for traversing the directory, AES encryption, Base64 encoding and deleting plaintext files.

Number of files	Total size (in bytes)	Time1	Time2	Time3	Time4	Average time
42	2.8MB	3s	1s	2s	1s	1.75s
53	4.1MB	5s	1s	3s	1s	2.5s
74	6.5MB	3s	2s	7s	1s	3.25s
112	10.2MB	11s	10s	6s	5s	8.0s

Table 5: The performance of encrypting USB profiles.

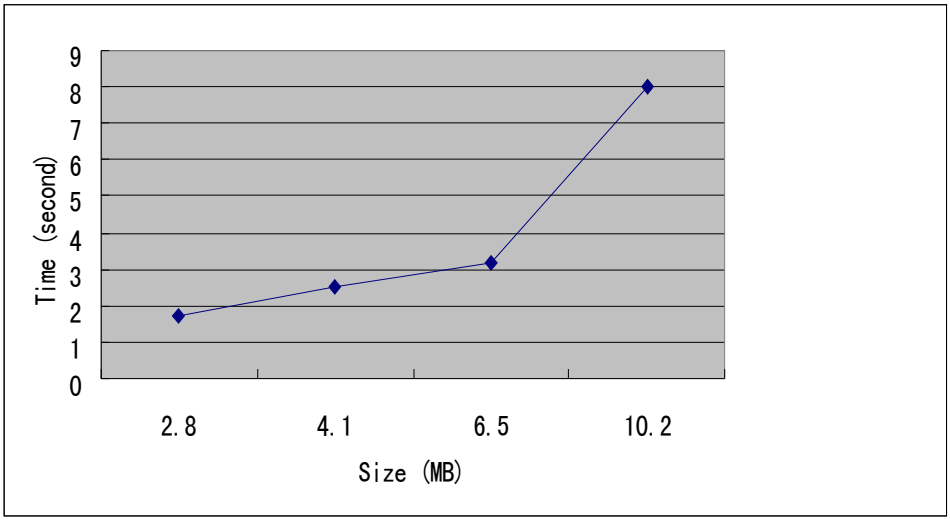


Figure 22: The elapsed time for encrypting profiles of different sizes.

From this table, it is worth mentioning that the time varies much from tests to tests. This is due to the fluctuation of the memory and CPU usage, and the content of the system cache as well. For small profiles, this variance is more obvious.

Typically users do not switch between profiles frequently. So some overhead should be acceptable in most cases. During the usability test, some users suggested that

a progress bar be displayed while doing encryption or decryption with a text explaining what is being done currently. Another suggestion is to allow users to decide which files to encrypt. For example, it can be added to the "Preferences" dialog, or a part of the "Profile Selection" dialog. The user can select to protect only the Mail folder or both the Mail and the prefs.js file. We can also choose to delete long-lived and untouched cache files instead of encrypting them.

V. CONCLUSION

The existing Mozilla profile manager allows users to customize their own settings of the browser. It provides great user interface and centralized yet flexible profile file location so that users can easily access, examine, modify the content. However, if a regular user can do so, so can any unauthorized user, especially when the profiles are stored on a removable disk. Also, the current profile manager does not achieve location independence because profiles cannot be loaded to a different workstation. This means that basically profiles created on one computer can only be used with that particular computer.

In this project, we solved these two problems by implementing the USB profile manager as an extension of the existing profile manager. The USB profile manager can automatically detect, load, and unload profiles stored on a USB key. Also, encryption, decryption and user authentication prevent unauthorized users from using or gathering

information from protected profiles. So the USB profile achieves both mobility and security.

Finally, the USB profile manager is a pure event-driven XPCOM component that can be easily installed or uninstalled without affecting other components.

According to our performance experiment, if the size of a profile is not very large, the actual time for encryption is acceptable given the fact that users do not switch profiles frequently.

VI. FUTURE WORK

Some future work has been proposed in earlier sections. Here is a summary. First, the *UpdateMountedUSBKeys()* function only works on Linux platforms. Work should be done to make this function completely platform-independent. Second, we need a more sophisticated way to determine whether a given directory is a valid Mozilla user profile, instead of just by checking the name of the directory. One approach is to check whether certain default files exist. However, this approach will add overhead. The third task is while loading a USB profile to the registry, also provide the version information. The last task is the provide a user interface to let the user decide which profile files to encrypt to further reduce the overhead.

ACRONYMS

AES – Advanced Encryption Standard

CBC – Cipher Block Chaining

DES - Data Encryption Standard

DOM – Document Object Model

ECB – Electronic Code Book

IV – Initialization Vector

NSS – Mozilla's Network Security Services

PSM – Mozilla's Personal Security Manager

USB – Universal Serial Bus

XPCOM – Cross Platform Component Object Model

XUL – XML User Interface Language

REFERENCES

[BC03] D. P. Bovet, M. Cesati. Understanding the Linux Kernel. O'Reilly. 2003.

[CR04] W. Chang, B. Relyea. "Network Security Services (NSS)". Retrieved on 4/2/04, from http://www.mozilla.org/projects/security/pki/nss/.

[04] A. Flett. “Guide to the Mozilla string classes”. Retrieved on 8/12/04, from http://www.mozilla.org/projects/xpcom/string-guide.htm.

[HWW02] P. Le Hégaret, R. Whitmer and L. Wood. “W3C Document Object Model.” July 17, 2002. Retrieved on 8/2/04 from http://www.w3.org/DOM/.

[KPS02] C. Kaufman, R. Perlman and M. Speciner. Network Security: Private Communication in a Public World. Prentice Hall. 2002.

[KR02] V. Klíma and T. Rosa. “Strengthened Encryption in the CBC Mode.” May 24, 2002. Retrieved on 9/24/04 from http://eprint.iacr.org/2002/061.pdf.

[LDH04] B. Lord, J. Delgadillo, T. Hayes. "Personal Security Manager (NSS)". Retrieved on 4/2/04, from http://www.mozilla.org/projects/security/pki/psm/.

[M03] Nigel Mcfarlane. Rapid Application Development with Mozilla. Prentice Hall. 2003.

[M98] The Mozilla Organization. “Embedding API Reference”. Retrieved on 8/2/04, from http://www.mozilla.org/projects/embedding/embedapiref/embedapiTOC.html.

[M98] The Mozilla Organization. "NSPR Reference". Retrieved on 4/2/04, from http://www.mozilla.org/projects/nspr/reference/html/index.html.

[M01] The Mozilla Organization. “Posing Gecko dialogs in embedding applications”. Retrieved on 8/2/04, from http://www.mozilla.org/projects/embedding/windowAPIs.html.
The Mozilla Organization. “Mozilla Cross-Reference”. From http://lxr.mozilla.org/seamonkey/.

[N98] Netscape Communications. “Gecko DOM Reference”. Retrieved on 8/2/04, from http://www.mozilla.org/docs/dom/domref/.

[P01] Rick Parrish. "XPCOM". Retrieved on 4/2/04, from http://www-106.ibm.com/developerworks/webservices/library/co-xpcom.html#h0.

[P99] B. Preneel. "State-of-the-art ciphers for commercial applications". Computers & Security. 1999.

[S95] B. Schneier. Applied Cryptography: Protocols, Algorithms and Source Code in C. Wiley. 1995.

[S99] W. R. Stanek. Mozilla Source Code Guide. Netscape Press. 1999.

[TO03] D. Turner, I. Oeschger. "Creating XPCOM Components". Retrieved on 1/12/04, from http://www.mozilla.org/projects/xpcom/book/cxc/.