PERSEUS: A Coding Theory-based Firefox Plug-in to Counter Botnet Activity

Eddy Deligne and Eric Filiol
ESIEA
Laboratoire de virologie et de cryptologie opérationnelles
38 rue des dr Calmette et Guérin
53000 Laval, France
`{deligne,filiol}@esiea-ouest.fr`

September 8, 2010

Abstract

Most of the activity of botnets is based on the ability to listen and analyze HTTP streams to retrieve and collect sensitive data (email address, login/password, credit card numbers...). This paper present an operational solution to counter botnets'activity through a simple Firefox plug-in. The core idea is to encode the HTTP traffic with variable punctured convolutional codes in such a way that any botnet client must face a time-consuming encoder reconstruction in order to decode. By adding noise in a suitable way, that reconstruction becomes untractable in practice and thus definitively hinders the botnet activity. On the users' side, encoder and noise parameters are first exchanged through an initial, short HTTPS session. The principles behind that approach have been mathematically validated in 1997 and 2007. The Firefox plug-in we present here has been developed under the triple GPL/LGPL/MPL licences. We present here its implementation and show that this encoding/decoding layer is fully transparent to the user and therefore does not degrade the overall performance contrary to any solution that would consider traffic encryption.

Keywords: Botnet - HTTP traffic - Firefox - Coding theory - Code reconstruction - Traffic eavesdropping.

1 Introduction

Most of the botnet's activity and payloads rely on listening and analyzing HTTP streams over the Internet. The aim is to collect sensitive data: email addresses for spamming, login/password for botnet clients further spreading, credit card numbers for carding... This is possible only because the HTTP protocol does not protect the content of transmitted packets. The use of encryption, besides the fact that it would lead to severe constraints (encryption overhead, key management...) poses problems in terms of legal regulations, especially in the context of transnational streams with respect to the different national regulations. Then the critical issue is: how can we protect against botnets'client wiretapping while allowing the action of States in the field of communication surveillance and while keeping the data transfer rate intact?

The project we are presenting in this paper aims at providing an operational solution to this issue. This solution is materialized in the form of a C++ Firefox plug-in named PERSEUS[1], developed under the triple GPL/LGPL/MPL licences and meeting the specifications of *Mozilla* development, thus allowing the code to be merged to the Firefox engine code directly.

Our approach rely on mathematical and coding theory principles which have been validated in 1997 [?], in 2001 [?] and in 2007 [?]. It mainly relies on research dealing with convolutional encoders reconstruction. In other words, how to reconstruct an unknown encoder to access to the data which have been encoded before the transmission?

The core idea is to encode the data exhanged (payload packets) with punctured convolutional codes. Those codes are commonly used in telecommunications (GSM, satellite...) due to their very high encoding speed. After this encoding layer and right before transmission, an artificial noise is applied to the data flow (as would any channel do). The noise is generated according to noise parameter $p = P[e_t = 1]$ where e_t is the noise bit at time instant t.

Now let us suppose that Alice wants to communicate with Bob over a HTTP traffic. As a first step, the different parameters of the variable encoder are randomly generated: polynomial size constraint, encoding rate, matrix punching, noise parameter p, encoder polynomials... Then a short HTTPS initial session allows to communicate those parameters to Bob (this amounts to about 256 bytes). Bob then will be able first to get rid of the artificial

[1]Perseus is the mythic heroes of Greek mythology who killed Gorgon Medusa. The botnets are thenselves often compared to Medusa and its long tentacles.

noise and then to set up the suitable Virberi algorithm for data decoding.

On the botnet agent side, the analysis of the HTTP stream must pass through a systematic preliminary phase of decoding. However since the encoder is changed whenever a new transmission occurs, the botnet agent must first reconstruct that unknown encoder as well as its different parameters. Since an artificial noise has been added to the encoded data, that reconstruction is known to be infeasible without heavy resources [?] which moreover would betray the presence of a botnet agent on any infected host. The time required for that reconstruction becomes quickly prohibitive even for reduced encoders. In addition, only an equivalent, non-punctured encoder can be recovered [?]. It is worth mentioning that if that reconstruction is beyond pratical capability of any botnet agent, it still remain tractable for any intelligence agency with a suitable computing power. Finally our experimental results show that our implementation is transparent to users and does not degrade the transmission performance.

This paper is organized as follows. Section 2 recalls basic facts about convolutional codes and their reconstruction. Section **??** presents the PERSEUS add-on structure while Section **??** deals with it detailed implementation. Section **??** presents the different experimental results we have obtained while Section **??** concludes by considering future evolution of this add-on.

2 Theoretical Background on Convolutional Codes

In this section, we are going to recall what a (punctured or not) convolutional code is as well as the main results with respect to their reconstruction. The aim is just to provide to the reader the required background to understand the interest of those codes and why they are particularly suitable for our approach.

2.1 Convolutional Codes

A convolutionnal encoder can be seen as an encoding system (based on a set of k shift-registers without feedback) such that, at each time instant, k information digits (typically the bits of packet payload) enter the encoder (one per register). Each information digit remains in the encoder for K time units and may affect each output during that time. The constant K is the constraint length or the *memory* of the encoder.

At each time instant, n information digits are output, each of them resulting from the XOR of k digits produced by the action of n polynomials on each register. The encoder is thus said to be of rate $\frac{k}{n}$. The action of the kn

polynomials and the shift are easily described by polynomial multiplications. So the polynomial representation will be used to represent the different streams.

A message will be composed of k interlaced input streams, each of them represented as a polynomial of degree $N+t$ denoted $a_i(x)$, $i = 1, \ldots, k$. The kn polynomials are of degree N (hence $N = K-1$) and will be noted $f_{i,j}(x)$. Then the encoder produces n output streams (of length t) represented as polynomials of degree t, $c_j(x)$, $j = 1, \ldots, n$ and we then have:

$$\sum_{i=1}^{k} a_i(x) f_{i,j}(x) = u_{j,1}(x) + x^N c_j(x) + x^{N+t} u_{j,2}(x) \tag{1}$$

The polynomials $u_{j,1}(x)$ (resp. $u_{j,2}$) (the filling (resp. the emptying) of the registers) are of degree at most $N-1$. Then the coded sequence is composed of the n interlaced output streams.

Thus the parameters of a convolutionnal encoder are:

- k and n defining the rate and the number of polynomials,
- K the constraint length (in fact it is related to internal memory of the encoder),
- the kn polynomials $f_{i,j}(x)$ of degree $N = K - 1$.

The convolutionnal encoder then describes a (n, k, N)-code. Generally, n and k are small integers with $k < n$. The most frequent case is $k = n - 1$. On the contrary, N must be made large enough to achieve low error probabilities. The symbols are usually elements of $GF(2)$ but generalization to $GF(q)$ where q is some prime power ($q = p^m$ for some positive integer m) can be easily done. We will only consider the case $q = 2$ but all the implementation and results can be generalized to any other prime q. This could be interesting in increasing the encoding speed.

Figure **??** describes a convolutional encoder of rate $\frac{1}{2}$.

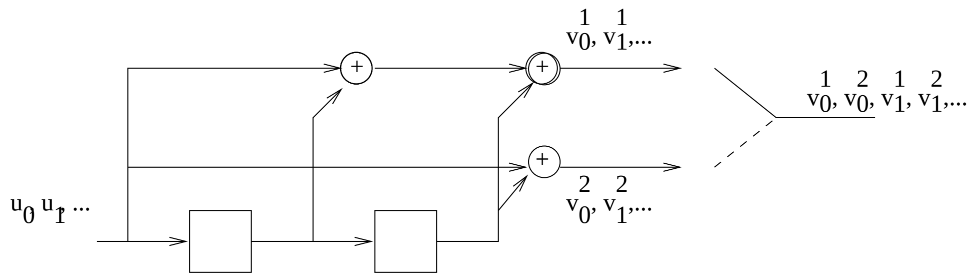

Figure 1: Convolutional encoder of rate $\frac{1}{2}$

In the context of PERSEUS, we will add an artificial noise of parameter p to the (encoded) output sequence $\mathbf{v} = v_0^{(1)}, v_0^{(2)}, v_1^{(1)}, v_1^{(2)}, \ldots$

The decoding step is performed through the classical Viterbi algorithm whose complexity is exponential in $k.N$. Hence, generally their use is limited to codes of short lengths and to reduced encoding rate $\frac{k}{n}$. However in our case since we completely master the noise (we exactly know where the noise bits are applied while any botnet agent does not), we can work with far higher value.

2.2 Punctured Convolutional Codes

Punctured convolutional codes were introduced by Cain *et al.* [**?**] as a means of greatly simplifying both Viterbi and sequential decoding of high rate convolutional codes at the expanse of a relatively small performance penalty.

A punctured convolutional code $\mathcal{C}$ is obtained by periodically deleting output symbols from a (base) (n, k, N)-convolutional code $\mathcal{C}_b$. Output symbols from $\mathcal{C}_b$ are deleted according to a periodic puncturing pattern (or perforation pattern) which can be described by its punctured matrix:

$$P = \begin{bmatrix} p_{1,1} & \cdots & p_{1,M} \\ \vdots & & \vdots \\ p_{n,1} & \cdots & p_{n,M} \end{bmatrix}$$

A very important problem is that of the reconstruction of such codes. In an attack context, a monitor wants to have access to the transmitted information (*the message*) without any knowledge on the encoder which produces the intercepted stream (*the coded sequence*). The only way is to reconstruct the encoder, that is to say to recover all its parameters. A simple decoding then gives access to the message.

Let us consider a (n,k,N)-(base) convolutional code $\mathcal{C}_b$. A given puncturing pattern P is a $n \times M$ $0-1$ matrix with a total of I 1's and $nM - I$ 0's where $p_{i,j} = 0$ indicates that the i-th symbol of every branch in the j-th treillis section (of the treillis diagram of $\mathcal{C}_b$) is to be deleted.

Then the original code $\mathcal{C}_b$, after being punctured with pattern P, has become a (I, kM, m)-(punctured) code [2] $\mathcal{C}$ [**?**].

Let us consider an illustrative, simple example.

Example 1 *Let us take the* $(2,1,3)$ *code with polynomials*

$$(1+x^2, 1+x+x^2)$$

The two output streams can be denoted as follows:

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \dots \\ y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \dots \end{pmatrix}$$

By using the following puncturing pattern:

$$P = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

we then obtain the two following output streams:

$$\begin{pmatrix} x_0 & & x_2 & & x_4 & & \dots \\ y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \dots \end{pmatrix}$$

that we can rearrange as follows:

$$\begin{pmatrix} x_0 & x_2 & x_4 & \dots \\ y_0 & y_2 & y_4 & \dots \\ y_1 & y_3 & y_5 & \dots \end{pmatrix}$$

It becomes then obvious that this puncturing produces a new encoder producing three output streams.

*By use of polycyclic pseudo-circulant matrices [**?**], the new parameters are easily defined and we have the 6 following polynomials*

$$f_{1,1}(x) = 1+x \quad f_{1,2}(x) = 1+x \quad f_{1,3}(x) = 1$$

$$f_{2,1}(x) = 0 \quad f_{2,2}(x) = x \quad f_{2,3}(x) = 1+x$$

where $f_{i,j}$ *denotes the j-th parity-check polynomial applied on input message stream* i.

As for Perseus is concerned, the puncturing pattern P will the last parameter to exchange during the initial HTTPS session.

[2] In fact, the degree of the punctured code may be less than N, but for most interesting punctured codes no degree reduction will take place

2.3 Reconstruction of Convolutional Codes

Since any punctured convolutional code is equivalent to a non punctured convolutional encoder, we will thus focus on the reconstruction of the latter codes. As far as code reconstruction is concerned, it is worth mentioning that the use of punctured codes make it more complex since we have equivalent non punctured codes whose parameters have higher values, for suitable values of I, k and M.

It is always possible to reconstruct convolutional codes in offline mode. This is basically not a problem since for most real cases, convolutional encoders do not change very often since they are hardwired (as an example, two convolutional encoders of constraint length of 9 are embedded in the UMTS standard [**?**]). Consequently we can spend a lot of time to reconstruct them since the work is done just once. However, there are only a very few known cases (most of them are for tactical, military communications like in the Czech army) where the encoders are randomly generated right before the transmission. The aim is clearly strongly hinders the code reconstruction which therefore cannot be online. In this latter case, except for very small values of parameters and noise probability, the reconstruction is too much time consuming.

The reconstruction of convolutional codes is a very mathematical stuff and consequently we will not present it here (see [**?**, **?**] for an exhaustive study). For our purposes, it is just necessary to recall the most significant results with respect to convolutional codes reconstruction.

While it is always possible to make the probability of false alarm (*i.e.* to reconstruct a wrong encoder) tend towards zero, the probability of success depends on many factors but the noise parameter has the most significant impact. Beyond 10 % the reconstruction will fail unless having a large amount of encoded sequence or/and accepting to spend a lot of time/machine ressources. In most practical cases, the Viterbi decoding itself is likely to fail for a few percent of noise (less than 0.05) long before the reconstruction process does. Expressing the reconstruction probability of success is not easy from a mathematical point of view and we advise the reader to refer to [**?**, **?**]. Experiments have confirmed that the reconstruction is bound to fail as soon as $p > 10\%$ unless spending a lot of time and computing power.

As for the computational complexity of the reconstruction, the general result [**?**, **?**] states that for a (n, k, N)-convolutional code, the lower bound is equal to $\mathcal{O}(\alpha \times n^5 \times N^4)$ where α is a constant which grows exponentially with the noise probability.

To illustrate that general result, Table **??** gives a few experimental results

[?, ?] for a few encoders in the case of a noise level of 10^{-2} and 2.10^{-2} (Gaussian noise).

Encoder	Reconstruction time ($p = 10^{-2}$)	Reconstruction time ($p = 10^{-2}$)
(4, 3, 8)	7 min 12 sec	Non detected
(4, 3, 9)	6 min 16 sec	Non detected

Table 1: Example of reconstruction time (on Pentium IV 2.0 Ghz) for two noise level

As a consequence, considering a rather high level of noise prevents the reconstruction to succeed unless we devote a huge computing time (several hours) which is far beyond the computing capability of any botnet client. We then will choose a noise level ranging from 0.25 to 0.35.

3 Presentation of the Perseus Firefox Add-on

The Firefox add-on Perseus aims at hindering botnets'ativities. It is written in C++, the native Mozilla Firefox language allowing for a possible incorporation into the code of Firefox. Therefore, the extension follows the principles of Mozilla coding style [?] and is fast, secure and multiplatform [?]. Perseus run on Firefox 3.0.3 and higher.

One main addon's goal is to make it completely transparent for the user when it is activated. The users will be continuing their web browsing without being aware of the underlying encoding process by Perseus. HTTP packets are then transmitted in an encoded, secure form. Mozilla Firefox has many useful functions for developing a plug-in, which are described in IDL interfaces. These interfaces are used with the tool *nsCOMPtr* that allows write code which is "shorter, cleaner, clearer and safer than that you can write with raw [XP]COM interface pointers". For more details on the interfaces and IDL nsCOMPtr see [?] and [?].

Perseus [3], behaves like a layer below the HTTP protcol as SSL/TLS does. The plug-in will deal of many tasks (see Figure **??**) :

1. intercept the HTTP requests. All HTTP requests are intercepted by Perseus, the interception implementation is explained in section **??**;

[3]Perseus means, from now and for the rest of the paper, our plug-in and no longer the mythological hero

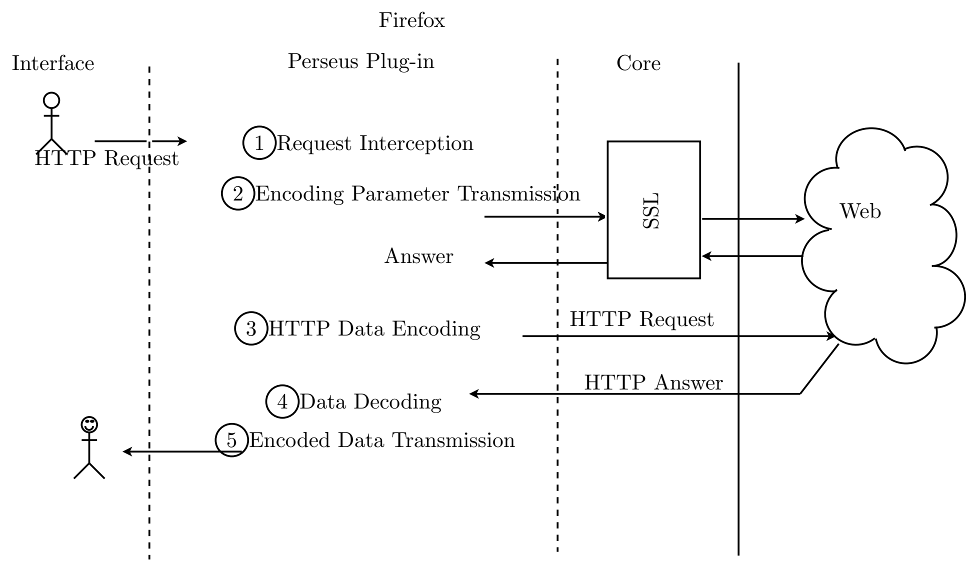

Figure 2: PERSEUS: general description.
"Interface" represents the GUI ; "PERSEUS plug-in" describes PERSEUS's characteristics ; "Core" is the rest of Firefox.

2. sends the encoder parameters. The encoder's parameters are generated by PERSEUS and next it sends them via HTTPS to the server (see section **??**);

3. encodes the data and sends them to the server. As soon as the server acknowledges receipt of encoder's parameters, PERSEUS encodes user's data contained in the HTTP request and sends them to the server (see section **??**);

4. intercepts the responses and decoding data, all HTTP request from the server to client, are analized by PERSEUS to verify that data are not encoded. If encoded data are detected, so then PERSEUS decodes them (see section **??**);

5. "sends" data to the user. The final stage consists in transmit the decoded data to Firefox so that it displays them.

3.1 Noise algorithm

According to the results presented in Section **??**, the application of noise to the encoded sequence, before transmission, prevents any practical reconstruction by any botnet client. Since this part is critical for the security (confidentility) of the HTTP packet payloads (against wiretapping), we are going to detail the mathematical and implementation aspects of the add-on devoted to the noise generation. First let us recall that if we denote $c = (c_0, c_1, c_2 \ldots)$ the convolutionally encoded sequence, then introducing noise consists in bitwise xoring the noise sequence $e = (e_0, e_1, e_2 \ldots)$. In other words we transmit the sequence

$$c \oplus e = (c_0 \oplus e_0, c_1 \oplus e_1, c_2 \oplus e_2 \ldots)$$

where $0.25 \leq p = P[e_t = 1] leq 0.35$.

Two secret parameters are used to define and computer the noise sequence. They are exchanged from the client to the server during the initial HTTPS session along with the other encoder parameters. Consequently, while the emitter and the recipient are always able to compute the precise indices of the noisy encoded bits and then to remove them before decoding (for the recipient who shares the two secret parameters with the emitter), any botnet agent cannot.

Upon reception of the noisy encoded sequence, the server removes the noise (compute the noise sequence and applies it to the received noisy encoded sequence relying on the fact that the XOR is an involutive operation) and then decodes the data without errors. The two secret parameters are X_0 and j.

- $X0$ is a random 63-bit integer,
- j is a random integer which vary between 0 and 10.

The integer X_0 enables to initialize the primitive polynomial P.

$$P = x^{63} + x^{15} + x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^{9} + x^{8} + x^{7} + x^{6} + x^{5} + x^{4} + x^{3} + x^{0}$$

j allows to select one number in our table *index*. This 64-bit integer contains between 25 % to 35 % of 1.

With those four parameters (X_0, P, j and *index*) we add the noise to the stream.

In Listing **??**, the noise is added to the *stream*, the polynomial P is given in its hexadecimal form. The function *random* (Listing **??**) produces a random 63-bit integer. With this number, the function *noise* (Listing **??**) creates a bit Et which represents the noise at time instant i.

```
void addNoise(char *stream,int stream_length,
                  long long int X0, unsigned int j)
{
    long long int reg = X0;
    /*P is a polynomial in hexadecimal*/
    long long int P = 0x800000000000FFF9;
    /* Et is a boolean which represent the noise*/
    int Et;
    for(int i=0; i<stream_length; i++) {
        /*reg is a 63 bits random integer*/
        reg = random(P,reg);
        /*noise generates 1 bit which has 30% chance to be a one*/
        Et = noise(reg,j);
        /* the noise is applied to the stream */
        stream[i] ^= Et;
    }
}
```

Listing 1: Noise function

By means of the polynomial P and reg (which depends on $X0$), a random integer is computed using a structure of linear feedback shift register of degree 63 whose output will be then vectorially filtered by the underbalanced Boolean function described by the integer $index[j]$ (Figure **??**). Those functions are given in Appendix **??**.

```
long long int random(long long int P, long long int reg)
{
    int rebouclage;
    for(int i=0; i<64; i++) {
        /*sumbit compute the sum of bits in GF(2) */
        rebouclage = sumbit(reg & P);
        reg >>= 1;
        reg |= rebouclage<<63;
    }
    return reg;
}
```

Listing 2: Random function

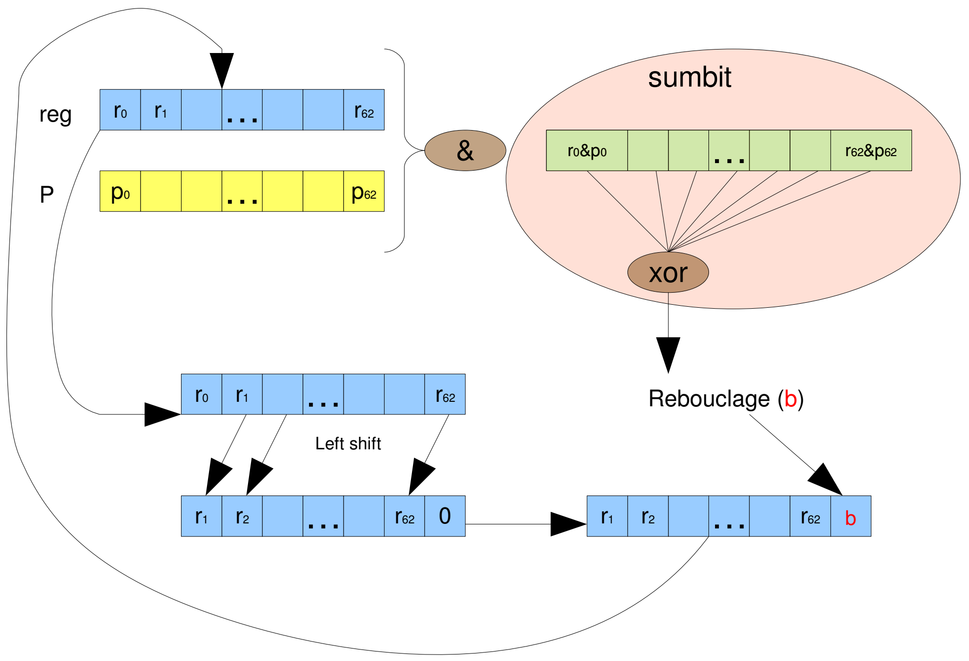

Figure 3: One rotation of *random* function

The *sumbit* function optimally computes the w sum of bits in $GF(2)$.

```
int sumbit(long long int w)
{
    w ^= w>>32;   w ^= w>>16;
    w ^= w>>8;    w ^= w>>4;
    w ^= w>>2;    w ^= w>>1;
    return (w & 0x1);
}
```

Listing 3: Sumbit function

The *noise* function select one bit in the integer *index[j]*.

```
int  noise(long long int reg, int j)
{
    /* 0 <= I <= 63 */
    int I = (reg>>33) & 0x3F;
    return ((index[j]>>I) & 0x1);
}
```

Listing 4: Noise function

4 Implementation of the Perseus Firefox Add-on

In this section, the different significant parts of the code source are detailed. Those parts allow to interact with Mozilla Firefox. Encoding and decoding are not explained since those algorithms follow the principles explained in Section 2 and do not use the Mozilla IDL interfaces. Moreover, we do not present the server part which is in fact a very classical *Apache* module we have specifically written for our module. The interested readers will find its code into the Perseus source code.

4.1 Intercept the web request

When the user accesses a site, Firefox creates a HTTP channel grouping all informations relating to the request (server address, port, protocol, etc.). To intercept the request, Firefox has a observer mechanism [**?**]. The observers are characterized by a "subject", the subject describes the action that must be observed. In Firefox 3.5, many subjects exist; here for our purposes, we focus on those related to the HTTP protocol.

1. `http-on-modify-request`: called as soon as a HTTP request is made;
2. `http-on-examine-response`: called after a response has been received from the webserver;
3. `http-on-examine-cached-response`: called instead of `http-on-examine-response` when a response will be read completely from the cache.

Observers are notified whenever those actions are occuring and the HTTP channel is transmitted as an argument. Perseus implements two observers, one to modify the HTTP requests and a second to modify the HTTP responses from webserver (listing **??**).

```
/** More information about ObserverService :
 *      http://mxr.mozilla.org/mozilla-central/source/xpcom/ds/nsIObserverService.
              idl
 */
nsresult rv ;
nsCOMPtr<nsIObserverService> obsSvc = do_GetService("@mozilla.org/observer-service;1
              ", &rv);
NS_ENSURE_SUCCESS(rv, rv); //Returns return-value if NS_FAILED(nsresult) evaluates
              to true, and shows a warning on stderr in that case.
/* Registers a given listener for a notifications regarding the specified topic.*/
rv = obsSvc->AddObserver(this, NS_HTTP_ON_EXAMINE_RESPONSE_TOPIC, PR_FALSE);
/** this : The interface pointer which will receive notifications
 *  NS_HTTP_ON_EXAMINE_RESPONSE_TOPIC:The notification topic.
 *  PR_FALSE : see the documentation
 */
```

```
NS_ENSURE_SUCCESS(rv, rv);

rv = obsSvc->AddObserver(this, NS_HTTP_ON_MODIFY_REQUEST_TOPIC, PR_FALSE);
NS_ENSURE_SUCCESS(rv, rv);
```

Listing 5: Registering observers

The function *observer* will be notified by Firefox whenever one of both actions will take place.

```
/** More information about Observer :
 *      http://mxr.mozilla.org/mozilla-central/source/xpcom/ds/nsIObserver.idl
 */
NS_IMETHODIMP
nsPerseusObserver::Observe(nsISupports *aSubject,
                            const char *aTopic,
                            const PRUnichar *aData)
{
    ...
    /* If a request is received */
    if (!strcmp(aTopic, NS_HTTP_ON_EXAMINE_RESPONSE_TOPIC)) {
        ...
    }
    /* If a request is sent */
    if (!strcmp(aTopic, NS_HTTP_ON_MODIFY_REQUEST_TOPIC)) {
        ...
    }
    ...
}
```

Listing 6: Processing notifications

For each request intercepted, the function will check:

- that the protocol is HTTP. HTTPS, ftp, file, etc. are not supported;
- that the request contains some data, with the method *POST* or *GET*.

Once cleared, the encoding parameters are generated and sent to the server.

4.2 Sending encoder's parameters to the server

4.2.1 Punctured convolutional code (PCC) parameters

Punctured convolutional code parameters are created randomly, we create new parameters for each new connection. This parameters are saved in a hash table with the server address used as a key. Firefox uses the NSPR[4] API for its IDL interfaces. This API has a class management hash table [**?**].

[4]The *Netscape Portable Runtime*.

```

/*PLHashTable *PL_NewHashTable(
   PRUint32 numBuckets,
   PLHashFunction keyHash,
   PLHashComparator keyCompare,
   PLHashComparator valueCompare,
   const PLHashAllocOps *allocOps,
   void *allocPriv
);*/

mTable = PL_NewHashTable(0, PL_HashString,
                          PL_CompareStrings,PL_CompareStrings,
                          &HashAllocOps,0);
```

Listing 7: Hash table creation

Our hash table uses integrates NSPR functions to hash the key (*PL_HashString*) and to compare it to the value (*PL_CompareStrings*). The `HashAllocOps` points to a structure which manages memory creation and destruction. The values saved in the table are PCC classes and their allocation and destruction we must done by the PCC class.

We can save and access the PCC code through the address server with the functions: *PL_HashTableAdd* and *PL_HashTableLookup*.

```
PLHashEntry *PL_HashTableAdd(PLHashTable *ht, const void *key,void *value)
```

```
void *PL_HashTableLookup(PLHashTable *ht, const void *key)
```

4.2.2 Send parameters

For the data transmission, a request is created using the `nsIIOService` [?]interface, which builds a HTTP channel with a URL, the protocol is determined by the scheme of this URL.

```
nsCOMPtr<nsIIOService> io = do_GetService("@mozilla.org/network/io-service;1");
nsCOMPtr<nsIChannel> channel;

NS_NAMED_LITERAL_CSTRING(url, "https://www.foo.com/");
nsCOMPtr<nsIURI> uri;
/*NewURI() : constructs a new URI by determining the scheme of the URI spec*/
rv=io->NewURI(url,nsnull,nsnull,getter_AddRefs(uri));
/*NewChannelFromURI() : Creates a channel for a given URI*/
rv=io->NewChannelFromURI(uri,getter_AddRefs(channel));
```

Listing 8: Creation of a request

Once the application has been created, the following parameters are set up:

- the mime type : ”perseus-init”;
- the sending methode : ”Post”.

The data are then injected in the request using the `nsI*Stream` stream interfaces [?]. Then the request is sent in synchrone mode[5] to the server. The data are sent securely with the protocol SSL/TLS using the URL scheme: HTTPS.

```
nsCOMPtr<nsIStringInputStream> stringStream = do_GetService(
              NS_STRINGINPUTSTREAM_CONTRACTID, &rv);
/*SetData() : assign data to the input stream*/
stream->SetData("toto",PL_strlen("toto"));

nsCOMPtr<nsISeekableStream> seekableStream = do_QueryInterface(stringStream, &rv);
/*Seek() : moves the stream offset*/
rv=seekableStream->Seek(0,0);

nsCOMPtr<nsIInputStream> inputStream= do_QueryInterface(stringStream, &rv);
nsCOMPtr<nsIUploadChannel> uploadChannel = do_QueryInterface(channel, &rv);
NS_NAMED_LITERAL_CSTRING(mimeType, "perseus-init");
/*SetUploadStream() : sets a stream to be uploaded by this channel*/
uploadChannel->SetUploadStream(inputStream,mimeType,-1);

nsCOMPtr<nsIHttpChannel> httpchannel = do_QueryInterface(channel, &rv);
NS_NAMED_LITERAL_CSTRING(methode,"POST");
rv=httpchannel->SetRequestMethod(methode);

nsCOMPtr<nsIInputStream> receiveStream;
/*Open() : synchronously open the channel*/
channel->Open(getter_AddRefs(receiveStream));
```

Listing 9: Send data

Once the server response is received, it must contain the status code number 200, which means that the request was successfully processed by the server. As well as the header “`Perseus:ack`” which allows to specify that the code has been understood and recorded by the webserver. If the checking phase is successful then the data are encoded, otherwise the data are sent unencoded.

4.3 Encoding information and sending it to webserver

Before encoding the data, we must extract the payload of the HTTP frames.

[5]The synchrone mode is prefered to the asynchrone one, despite its blocking character; the webserver response enables to determine whether the server supports PERSEUS or not.

4.3.1 Extracting data

The data can be sent out by two method : *POST* or *GET*.

- The ***GET*** method allows to insert data into a URL, eg "http://www.foo.com/bar.php?login=toto?pass=tata", contains information about login and password for the server *foo.com*. If this method is detected, we must extract the data (listing **??**) that are in the *path* variable of the URL (in our example the *path* variable contains : "bar.php?login=toto?pass=tata").

  Only the part which follows the first question mark is extracted and encoded, the question mark allows to distinguish the data from the rest of the URL [**?**].

```
nsCOMPtr<nsIHttpChannel> channel = do_QueryInterface(aSubject, &rv);
nsCAutoString method;
rv=channel->GetRequestMethod(method);

if (method.Equals("GET")) {
    rv=uri->GetPath(data);
    NS_ENSURE_SUCCESS(rv, rv);
    if(data.IsEmpty()) {
        return NS_OK;
    }
    if (data.FindChar('?') != -1) {
        data.Cut(0,data.FindChar('?')+1);
    }
    else return NS_OK;
```

Listing 10: Data extraction for *get* method

- In the ***POST*** method, data are included in the HTTP body, so we must use another mechanism to extract data. From the HTTP channel, we get the service that sends data: the *uploadChannel*[**?**] interface. Through this interface, we get the stream which contains the data. We also must discriminate two different data types: one has a known mime type while the second one has raw data type (listing **??**).

```
nsCOMPtr<nsIUploadChannel> uploadchannel = do_QueryInterface(aChannel,&rv);
nsCOMPtr<nsIInputStream> inputstream;
rv=uploadchannel->GetUploadStream(getter_AddRefs(inputstream));
nsCOMPtr<nsIMIMEInputStream> mimeStream = do_QueryInterface(inputstream,&rv);
if(NS_SUCCEEDED(rv)) {
    \\Read data with mime type
}
nsCOMPtr<nsIStringInputStream> stringStream = do_QueryInterface(inputstream,&
                rv);
if(NS_SUCCEEDED(rv)) {
    \\Read raw data
}
```

Listing 11: Data extraction for *post* method

If data have a mime type then the stream will contain the mime type at beginning of the flow (preceded by "Content-type" and the data length (preceded by "Content-length"); whereas the mime type and the data length are included in the HTTP header for the raw data.

Example 2 (Data with a mime type)

```
Content-type:application/x-www-form-urlencoded
Content-length:25
login=toto&password=tata
```

Example 3 (Raw data)

```
login=toto&password=tata
```

Depending on the case, data are extracted then encoded, the mime type remains unchanged and the data length is computed with new encoded data. The HTTP frame is then reconstructed using the two methods previously stated.

We add the header "`Perseus : pcc`" to the HTTP request in order to discriminate an encoded content from a nonencoded one. The encoding technique is specified as the header value, allowing future features to use more encoding algorithms.

Firefox send the request to the webserver, at the end of the manipulation on the HTTP frame (i.e. leaving the function *Observe* (listing **??**)).

4.4 Interception and decoding responses

Through the observer "`http-on-examine-response`" (listing **??**), the plugin analyzes each response from the webserver by the function *Observe* (listing **??**) when it is switched on.

Responses with the headers "`Perseus : pcc`" (representing a response encoded with a punctured convolutional code) are processed, others are discarded and sent directly to Firefox for display.

When the observer is notified, only headers are available, the rest of the HTTP frame is not downloaded yet. To get data, firefox 3.0.3 introduce a new interface *nsITraceableChannel* [**?**] which enables to modify the listener[6]. We must write a class which implements a *nsIStreamlistener* [**?**] to acquire data, then decodes them and restore them to Firefox.

```
PerseusDownloadData *down = new PerseusDownloadData ;
/*PerseusDownloadData must implement nsIStreamListener */

nsCOMPtr<nsITraceableChannel> trace = do_QueryInterface(httpchannel, &rv);
/** SetNewListener() replaces the channel's current listener with a new one,
 *  returning the listener previously assigned to the channel
 *  The previously listener is saved in mListener.
 */
rv= trace->SetNewListener(down,getter_AddRefs(down->mListener));
```

Listing 12: Modification of the "listener"

The class implementing the interface *nsIStreamListener* must embed three functions:

1. *onStartRequest*: a request begin. Variables are initialized (buffer, octet read);

2. *onDataAvailable*: data are reachable and saved into the buffer;

3. *onStopRequest*: the end of the request. Data are processed (i.e. decoded).

```
 1  NS_IMETHODIMP
 2  PerseusDownloadData::OnStartRequest(nsIRequest* aRequest,
 3                                      nsISupports* aContext)
 4  {
 5    nsresult rv;
 6    mRead=0; //number of bytes read
 7    mData=0; //buffer
 8    /*We pass the request to the previous listener.*/
 9    mListener->OnStartRequest(aRequest,aContext);
10    return NS_OK;
11  }
```

Listing 13: OnStartRequest : variables initialization

In Listing **??**, we find buffer's initialization (*mbuffer*) in which we save incoming data, and as well as the variable's reset (*mRead*) which allows to count characters' number. At line 8, the request is proceeded to the old listener, saved when we update "listener".

```
S_IMETHODIMP
PerseusDownloadData::OnDataAvailable(nsIRequest* aRequest,
```

[6]A "*listener*" is a function that listen data arrival.

```
                             nsISupports* aContext,
                             nsIInputStream *aIStream,
                             PRUint32 aSourceOffset,
                             PRUint32 aLength)
{
   /**If the stream is more longer than the http protocol support then the
    * function OnDataAvailable will be run many times.
    */
   nsCOMPtr<nsIBinaryInputStream> binaryInStream =
      do_CreateInstance("@mozilla.org/binaryinputstream;1", &rv);
   rv=binaryInStream->SetInputStream(aIStream);

   mData= (char *) PR_Realloc((void *) mData,sizeof(char) * (aLength+mRead) );
   if (!mData) {..}

   char * data = (char *) PR_Malloc(sizeof(char)*aLength);
   rv=binaryInStream->ReadBytes(aLength,&data);
   /*We read aLength octet from the aIStream*/
   memcpy(mData+mRead,data,aLength);
   mRead+=aLength;
   free(data);
   return NS_OK;
}
```

Listing 14: onDataAvailable : capture data

The listing **??** show the data recording. The function *OnDataAvailable* can be executed many times if the request is cut in many pieces. So we must reallocate the buffer (*mbuffer*) to the length of data read plus the length of data which are going to read.

```
NS_IMETHODIMP
PerseusDownloadData::OnStopRequest(nsIRequest* aRequest,
                                nsISupports* aContext,
                                nsresult aStatus)
{
   /*Here we decode data -> datadecoded is create*/
   nsCOMPtr<nsIStorageStream> storageStream = do_CreateInstance(
                 NS_STORAGESTREAM_CONTRACTID, &rv);
   rv=storageStream->Init(8192,mRead,nsnull);
   nsCOMPtr<nsIBinaryOutputStream> binaryOutStream =
      do_CreateInstance("@mozilla.org/binaryoutputstream;1", &rv);
   nsCOMPtr<nsIOutputStream> out;
   rv=storageStream->GetOutputStream(0,getter_AddRefs(out));

   rv=binaryOutStream->SetOutputStream(out);
   rv= binaryOutStream->WriteBytes(datadecoded.get(), datadecoded.Length());

   nsCOMPtr<nsIInputStream> inputStream ;
   rv=storageStream->NewInputStream(0,getter_AddRefs(inputStream));

   /*We pass the decoded data to the previous listener.*/
   mListener->OnDataAvailable(aRequest,aContext,inputStream,0,datadecoded.Length());

   /*We pass the request to the previous listener.*/
```