MOMENTUM

Una guía sobre emulación de Habbo

Agradecimientos

Para empezar quiero agradecer a todas las personas que han prestado sus conocimientos para realizar este proyecto.

LittleJ - Gracias a sus conocimientos avanzados de criptografía, matemáticas y sobre todo su grandísima paciencia (sin mencionar que es uno de los padres de todo esto y de la new Crypto), la comunicación constante con LittleJ ha sido posible conocer mas sobre este mundo.

Kolesias - Por INIX ya que gracias a su invitación a inix y a su implementación de ciertas partes de su emulador he podido comprender muchas cosas sin él no habría sido posible nada de esto.

NeoForex - Por ayudar en la parte de comprensión del inicio de la comunicación con el cliente y el servidor.

Xenok - Ayudó mucho en el lenguaje y ha sido una guía en este proceso, además de que el inicio del proyecto Nostalgia se debe a su idea de crear un emulador con dicho nombre y en NodeJS (esto es un secreto espero que no se lo digan a nadie), **también ha creado la genial portada de la guía**, todo proyecto debería tener a un Xenok.

AlfonsoMV - Por leer el texto y corregir ciertas partes.

Robercid - Por decir que esto no se puede hacer con PHP.

Todas estas personas han colaborado en este proyecto ya que ellos en su momento han podido aprender mucho en este entorno y quieren compartir sus conocimientos para que otros puedan aprender o interesarse por todo esto. No hablamos de crear holos ni nada de eso, todos lo vemos desde un punto de vista didáctico y es interesante las matemáticas implicadas y las soluciones que han surgido a lo largo de todo este tiempo para poder realizar una comunicación exitosa, os animo que como estas personas han sabido aprender y compartir, vosotros en el día de mañana podías hacer lo mismo, espero que esta guía sea de vuestro agrado.

Prefacio

Todo tiene un inicio y este es el nuestro: hace mucho tiempo las personas que querían iniciar este viaje no tenían una guía hacia donde ir o como empezar su viaje, era un viaje casi a ciegas y muy difícil, solo se contaba con indicios de otros viajantes que ya habían logrado terminar este viaje exitosamente sin embargo las pistas que nos dejaron estos viajantes era muy difusa y de difícil lectura ya que estaba escrito en un lenguaje que nadie puede comprender (C#).

Eran tiempos difíciles, tiempo en donde la gente ya no innovaba tiempos en donde solo se hacían re-cocinados de otros proyectos, una vez y otra vez, re-cocinado y cambiando el nombre de la receta haciendo pasar esta receta como una nueva receta, ahora ha cambiado todo y para ese propósito se presenta esta guía de inicio.

¿A quiénes va dirigida la guía?

La guía va dirigida a los programadores que saben usar bien un lenguaje, tienen conocimientos sobre conversión de base (binaria a decimal, hexadecimal, etc…), que sepa crear sockets, y a los no encontraron la forma de crear un emulador desde cero ya que se hace imposible iniciar algo sin tener conocimientos de como funcional ese algo.

Va dirigida a esas personas que les gusta aprender y enseñar a los demás, a personas que no saben programar pero quieren aprender y en un futuro poder usar esta guía.

No va dirigida a las personas que solo quieren copiar y pegar algo ya creado previamente, a las personas que no quieren aprender y quieren seguir haciéndolo de siempre.

No va dirigida a las personas que ya saben todo esto y que son felices con sus conocimientos.

¿Qué es Momentum?

Momentum es una guía que pretende hacer una introducción hacia la emulación, no pretende ser una guía de estilos ni tampoco una base para un emulador profesional, tampoco pretende enseñar cómo se crackea un swf, (usaremos uno ya creackeado por la comunidad), **ni tampoco pretende ser un curso de programación**, simplemente pretende dar los primeros pasos a esos programadores que desconocen el comportamiento inicial de un emulador.

Abarcaremos varios temas relacionado con el inicio del emulador como son:

- Conocer los mensajes que nos envía el cliente en cada paso.
- Conocer los paquetes y trabajar mejor con ellos.
- El uso y creación de paquetes para la comunicación con el cliente.
- Comprensión de la técnica de comunicación (cifrado) con el servidor.

Todo lo que se tratará de explicar será necesario para lograr hacer el handshake[1] con el cliente.

[1] Handshaking es una palabra inglesa cuyo significado es apretón de manos y que es utilizada en tecnologías informáticas, telecomunicaciones, y otras conexiones. Handshaking es un proceso automatizado de negociación que establece de forma dinámica los parámetros de un canal de comunicaciones establecido entre dos entidades antes de que comience la comunicación normal por el canal. De ello se desprende la creación física del canal y precede a la transferencia de información normal. Fuente: WIKIPEDIA

Inicio

Para empezar con todo esto lo primero que tendríamos que saber es con que versión del swf vamos a trabajar y en esta guía se trabajará con esta versión: **PRODUCTION-201607262204-86871104**[2] *(esta versión ya está crackeada por la comunidad).*

Tener un entorno de trabajo, como lo son todas las herramientas que usas para la creación de tus geniales programas.

Para poder hacer funcionar las pruebas con el cliente necesitaremos un swf un cliente.html (con todas sus dependencias), y un servidor HTTP en donde irá el swf, el XAMPP nos iría de lujo, pero si decides usar otro pues no hay problema.

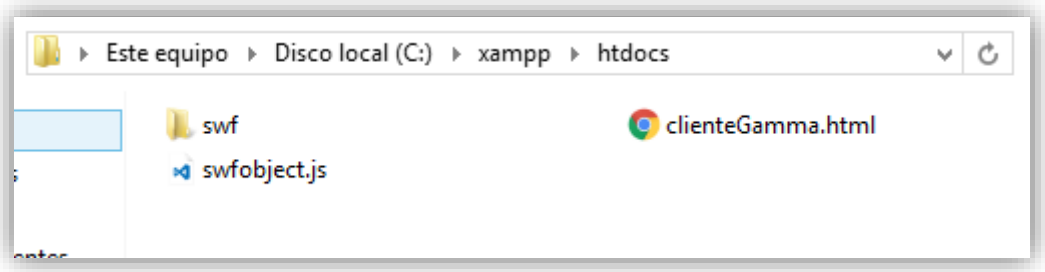
Para la descarga del cliente y el swf proporciono en este enlace:

DESCARGA ESTO, ES IMPORTANTE PARA EL PROYECTO:

https://mega.nz/#!0pE0RbbI!EfTP1AeOZ7aq6VWr2eEPcSXf2d5Mo8aI4llipyKJhKw

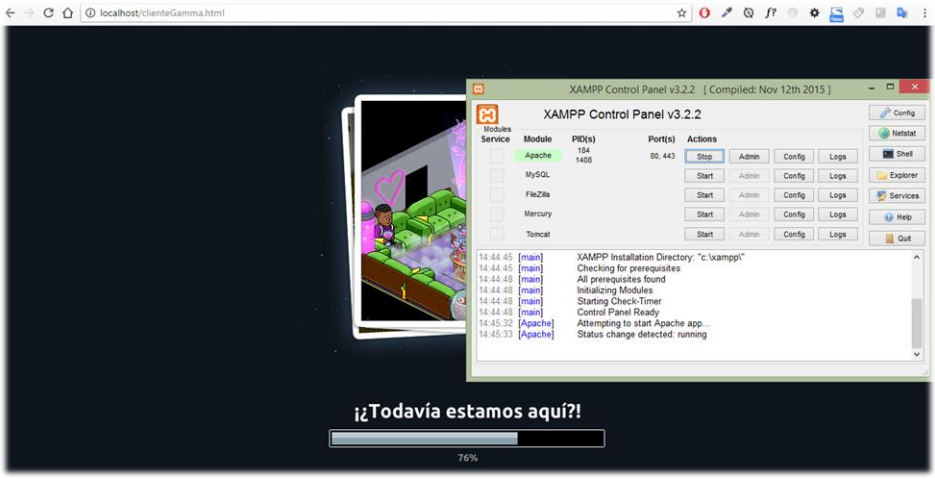
En el archivo que hemos descargado anteriormente contamos con todo lo necesario y configurado para hacer las pruebas.

Solo basta con descomprimir todo el .zip dentro de la carpeta htdocs (en caso de xampp) o una carpeta donde podamos ejecutar el servidor **HTTP**.



[2] Software en producción es un software que está abierto al público es decir que está en funcionamiento, en este caso vemos que después de la palabra "PRODUCTION" hay unos números, esos números significan la fecha, como por ejemplo "PROUCTION-2016…" es del año 2016.

El cliente lo cargaremos siempre con el XAMPP o con tu servidor HTTP favorito que tengas, a partir de acá ya tienes que tener tu cliente puesto en tu servidor y podrás acceder a el con “**http://localhost/clienteGamma.html”**.



También necesitaremos mucha confianza y ganas de trabajar ya que a lo mejor a la primera no se consigue pero créeme cuando te digo que al final lo conseguirás y cuando lo consigas sentirás que todos tus esfuerzos habrán valido la pena, veras que con cada paso de la comunicación te acerca más al objetivo y al final de todo esto serás una persona totalmente distinta.

El cliente

Una vez que hemos descargado el cliente lo abrimos con un editor de texto (yo uso Visual Studio Code) y editamos nuestro "**clienteGamma.html**", si nos fijamos ya he editado el cliente para que esté limpio y también le he agregado unas variables para que su configuración sea rápida.

```
<body>
    <div id="flash-container"></div>
    <script type="text/javascript">
        // Configurame aquí
        var config = {
            url: "localhost",       // URL de las SWF, localhost es la resolución a la ip 127.0.0.1
            emuUrl: "localhost",    // dirección IP del emulador, localhost es la resolución de la ip 127.0.0.1
            port: "3030",           // Puerto del emulador
            usuario: "gammafp",     // Usuario que se conecta
            auth: " "               // Autentificacion para sso.ticket
        };

        var flashvars = {
            "client.allow.cross.domain": "0",
            "client.notify.cross.domain": "1",
```

Y ¿qué es lo que hace el cliente realmente?, pues lo que hace es cambiar dinámicamente esas variables para cada usuario, exceptuando las variables **url**, **emuUrl** y **port**, estas no cambian.

Las variables de **usuario** y **auth** cambian dinámicamente con cada usuario, pero en el cliente que hemos creado las variables **usuario** y **auth** las tenemos fija para un solo usuario, solo para hacer las pruebas.

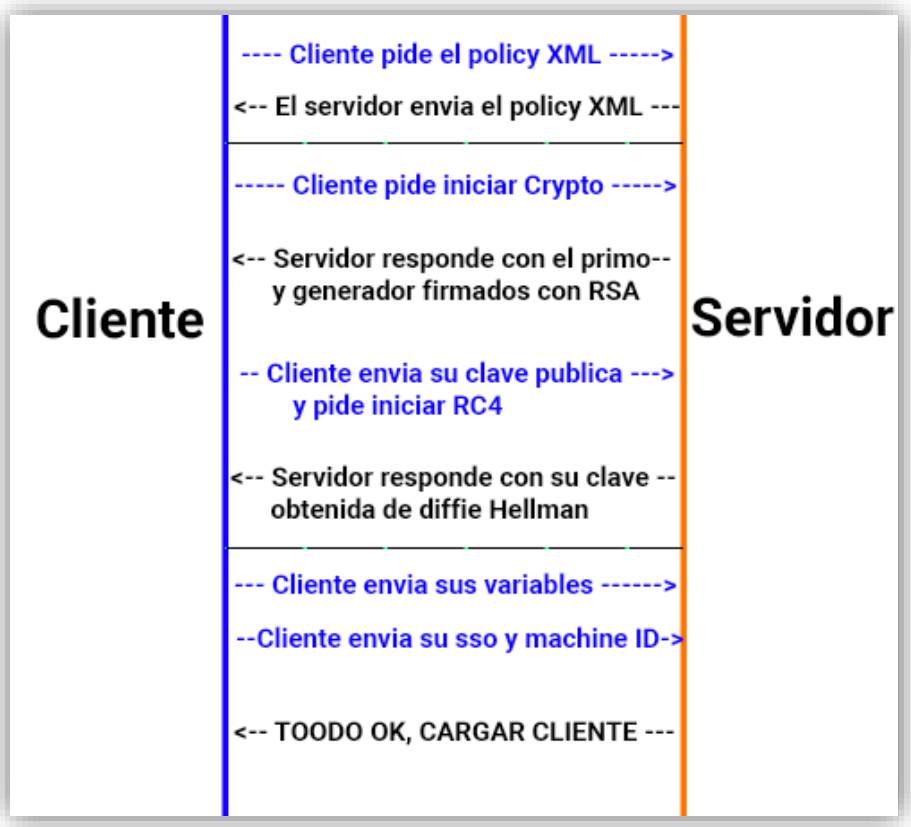
Listo ya tenemos nuestro cliente funcionando en **localhost**[3] y en el puerto **3030**, esto es importante tenerlo presente, con esa configuración el cliente va a buscar **localhost** y el puerto **3030** a eso se le denomina un **socket**.

El cliente cuando se ejecuta va a buscar el socket **127.0.0.1 (localhost)** y el puerto **3030** entonces nosotros tenemos que crear nuestro emulador con el mismo socket que el cliente.

[3] Para entendernos, en redes localhost sería como un dominio interno del pc el cual normalmente apunta a la dirección ip 127.0.0.1 en IPv4 y ::1 en IPv6, se usa localhost en lugar de una ip debido a que la ip local puede ser distinta a la 127.0.0.1 y con localhost el ordenador ya sabe que ip tiene como loopback local.

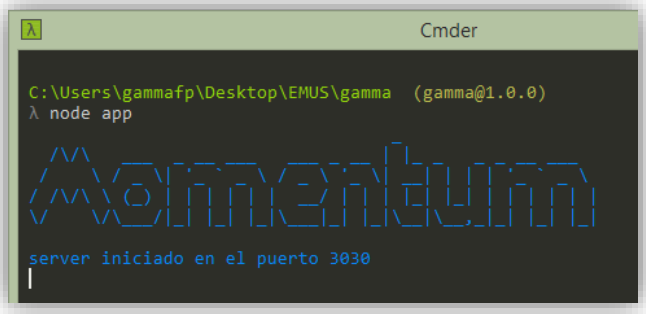
Primer acercamiento al emulador

Vamos a hablar que pasa entre el cliente la siguiente imagen podemos ver como es la comunicación entre el cliente y el servidor:



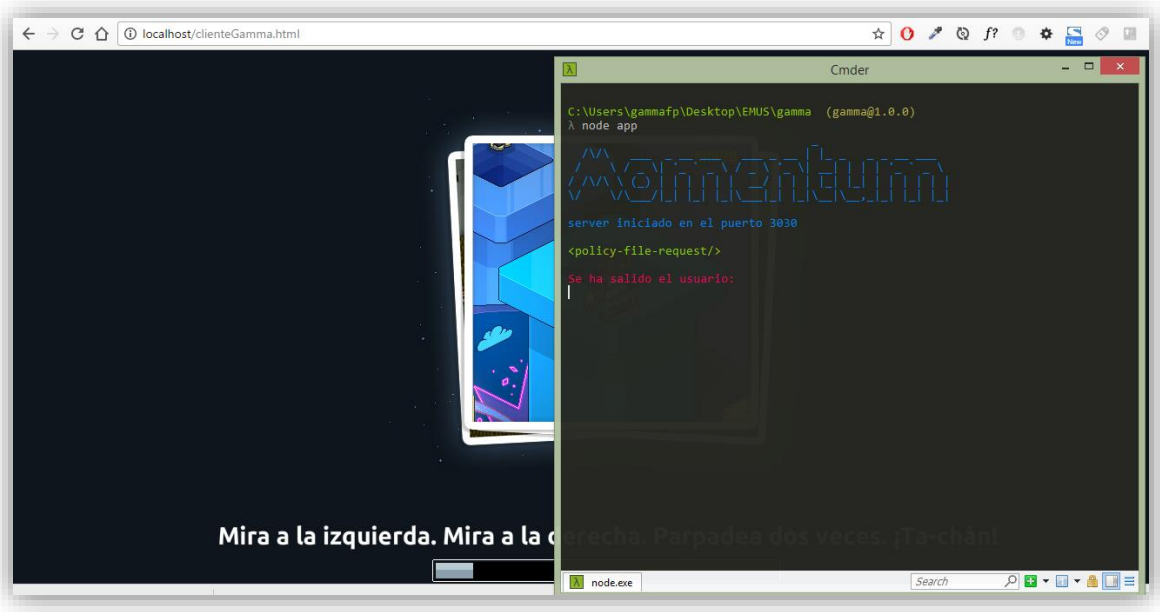
La comunicación que ocurre entre el cliente y el servidor se hace a través de un socket TCP/IP (si no sabes que es, deberías buscar antes la manera de crear un chat simple por socket en tu lenguaje favorito).

Primero crearemos nuestro servidor esperando a recibir algún mensaje:



Cliente envía la petición del policy

Emocionadísimos pensando que al iniciar el cliente ocurrirá un milagro y todo irá como la seda nos aparece una hermosa sorpresa:



Recordando la imagen del handshake podemos ver que en nuestra primera interacción el cliente nos pide un **policy-file-request** y como todo buen noob no sabrás que cosa es esa o con que magia negra se sigue el siguiente paso, pues el **policy-file-request** vendría siendo como permisos de dominios, es decir que sería como dar permisos a ciertos servidores a realizar las conexiones, y nosotros pues usaremos este de acá:

```
"<?xml version=\"1.0\"?>\r\n<!DOCTYPE cross-domain-policy SYSTEM \"/xml/dtds/cross-domain-policy.dtd\">\r\n<cross-domain-policy>\r\n<allow-access-from domain=\"*\" to-ports=\"*\" />\r\n</cross-domain-policy>\0"
```

Lo anterior es el policy, ese texto se envía tal cual está a nuestro cliente (mostraré dos ejemplos uno en node y otro en php):

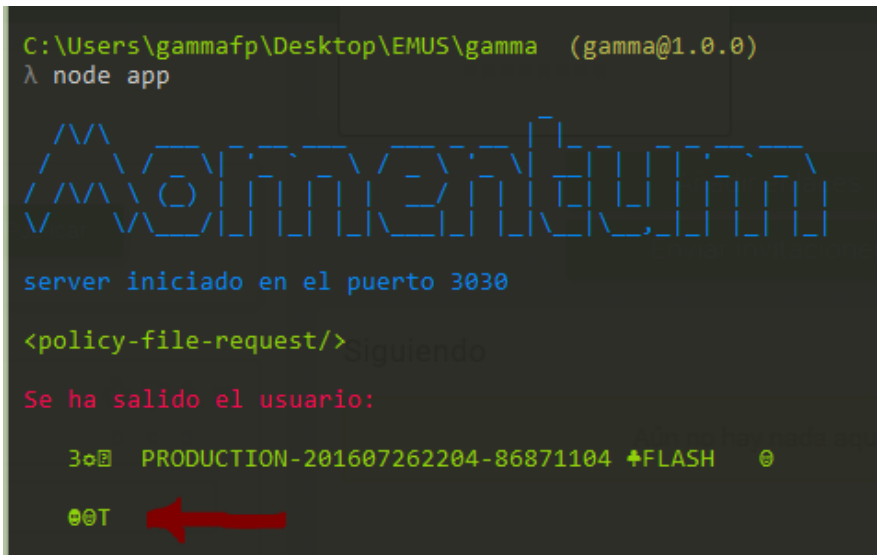
Ejemplo en NodeJS:

```
    if(data[0] === 60) {
        console.log("Enviando policy");

        socket.write("<?xml version=\"1.0\"?>\r\n" +
        "<!DOCTYPE cross-domain-policy SYSTEM \"/xml/dtds/cross-domain-policy.dtd\">\r\n" +
        "<cross-domain-policy>\r\n" + "<allow-access-from domain=\"*\" to-ports=\"*\" />\r\n" +
         "</cross-domain-policy>\0");

    } else {
```

Ejemplo en PHP:

```
if(strlen($mensaje) == 23) {
    echo "Se ha enviado el policy";
    $policy = "<?xml version=\"1.0\"?>\r\n" . "<!DOCTYPE cross-domain-policy SYSTEM \"/xml/dtds/cross-domain-policy.dtd\">\r\n" .
     "<cross-domain-policy>\r\n" . "<allow-access-from domain=\"*\" to-ports=\"*\" />\r\n" . "</cross-domain-policy>\0";
    socket_write($socket_new, $policy, strlen($policy));
}
```

Cuando enviamos nuestro querido policy-file-request recibimos un mensaje por parte del cliente.



El cliente nos envía su versión y también nos envía caracteres alienígenas dignos en salir en el canal de Mundo Desconocido, el caso es que esos caracteres hay que tratarlos, pero no contamos en este momento con los medios de lograrlo así que tendremos que crear una bonita librería para poder leer paquetes y poder interpretarlos, así que vamos a ello.

Cogemos un montón de paciencia y muchas ganas de trabajar para no rendirte, ya que esta parte es muy crucial para poder continuar nuestro inicio, vamos a ello.

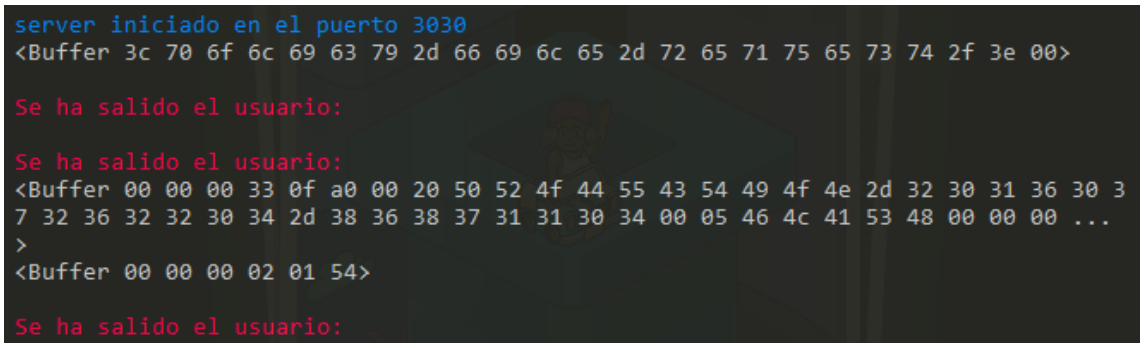
Creación de nuestro packet reader

Al inicio no sabemos como continuar en este punto, cuando nos encontramos con estos caracteres "☻☺T" no sabemos realmente que significan o que hacen o como continuar (si estas en algún lenguaje como nodeJS o Dart este mensaje puede ser representado en una lista o buffer de enteros de 8 bits)

Los caracteres antes mencionados guardan el mensaje que nos ha enviado el cliente, recordando la imagen que representa la comunicación entre el cliente y el servidor podemos observar que el siguiente paso que nos pide el cliente es **iniciar Crypto,** pero eso no lo podemos ver con esos caracteres raros y tampoco podemos identificar los pasos siguientes, para poder saber en que paso estamos necesitamos crear un lector de paquetes el cual separará esos caracteres raros y asi poder obtener un **HEADER** y en un **BODY**

El **HEADER** ayuda a identificar el evento que nos está pidiendo el cliente y también nos ayuda a enviar diferentes eventos al servidor, así que existen **HEADERS** tanto para cliente (que envía al servidor) y para servidor (que envía al cliente).

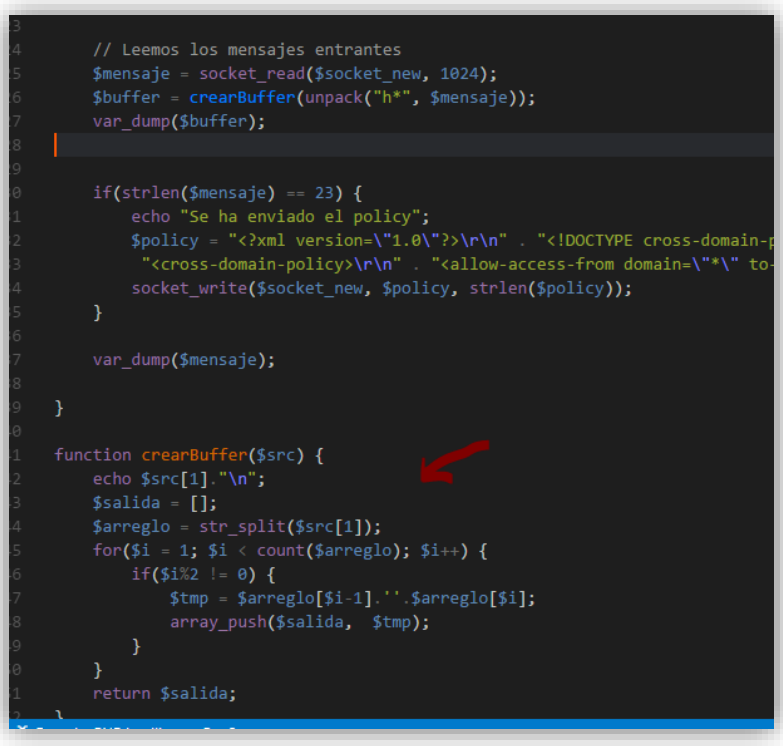
Si usas NodeJS o Dart tienes la suerte de que el mensaje es representado en un buffer de de ocho bits (sin signos):



La representación en buffer de 8bits sin signo de los mensajes anteriores, por ejemplo este buffer (00 00 00 02 01 54) representaría este mensaje: "☻☺T", **NOTA: ESTÁ CODIFCADO EN HEXADECIMAL, SIN EMBARGO SIGUEN SIENDO 8 bits HAY QUE TENERLO PRESENTE**

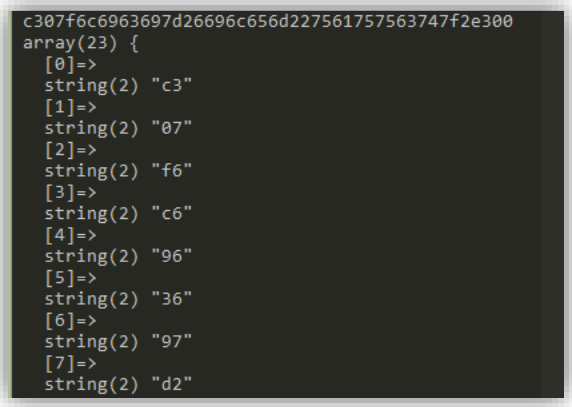
Pero si eres de otro lenguaje el cual es incapaz de representar ese mismo buffer, pues te recomendaría ver como puedes convertir la entrada del socket en un array/lista hexadecimal de 8 bits, lo digo por el hecho de que te será mas fácil seguir la guía.

En PHP yo hice algo parecido a eso, y me va perfecto para poder trabajar con los bits, así que podéis o no seguir mi ejemplo:



```
    // Leemos los mensajes entrantes
    $mensaje = socket_read($socket_new, 1024);
    $buffer = crearBuffer(unpack("h*", $mensaje));
    var_dump($buffer);


    if(strlen($mensaje) == 23) {
        echo "Se ha enviado el policy";
        $policy = "<?xml version=\"1.0\"?>\r\n" . "<!DOCTYPE cross-domain-p
         "<cross-domain-policy>\r\n" . "<allow-access-from domain=\"*\" to-
        socket_write($socket_new, $policy, strlen($policy));
    }

    var_dump($mensaje);

}

function crearBuffer($src) {
    echo $src[1]."\n";
    $salida = [];
    $arreglo = str_split($src[1]);
    for($i = 1; $i < count($arreglo); $i++) {
        if($i%2 != 0) {
            $tmp = $arreglo[$i-1].''.$arreglo[$i];
            array_push($salida,  $tmp);
        }
    }
    return $salida;
}
```

En la imagen anterior he usado **unpack** para poder convertir la entrada del socket a hexadecimal, luego le paso una función que separará el **string** hexadecimal a 8 bits y el resultado es el siguiente:



```
c307f6c6963697d26696c656d227561757563747f2e300
array(23) {
  [0]=>
  string(2) "c3"
  [1]=>
  string(2) "07"
  [2]=>
  string(2) "f6"
  [3]=>
  string(2) "c6"
  [4]=>
  string(2) "96"
  [5]=>
  string(2) "36"
  [6]=>
  string(2) "97"
  [7]=>
  string(2) "d2"
```

Si nos fijamos en la imagen con la función creada por mi da el mismo resultado que en NodeJS.

Recordemos un poco la teoría de conversión binaria:

En hexadecimal cada carácter abarca cuatro bits es decir que el carácter **F** va a ser igual 1111 y si agrupamos dos F pues el resultado serian ocho bits, FF = (1111 1111), teniendo esto presente podremos trabajar tranquilamente,

pero si a este punto no te acuerdas o no sabes nada de conversión binaria pues te recomiendo que leas al respecto, hay mucha información y viene bien recordar esto.

Imaginemos por un momento que el cliente nos manda un paquete (no el anterior, este paquete será para hacer el packet reader), y el paquete que nos manda es el siguiente:

```
[0,0,0,21,9,159,0,25,1,145,0,61,104,116,116,112,58,47,47,108,111,99,97,108,104]
```

Si ya sé, ¿pero no habías dicho que es mejor convertir socket en **HEXADECIMAL**? Yo ahí lo veo en **DECIMAL ¡me mientes!**

Tranquilidad, el paquete anterior en decimal es lo mismo como si fuese hexadecimal, recordemos que para poder operar en los números nos vendría bien convertirlos a decimales ya que nos es mas fácil, por suerte node ya te cambia la representación automáticamente, y si estáis con otro lenguaje como puede ser **PHP** pues tenéis que hacer una librería que cambie cada hexadecimal a su versión en decimal y viceversa.

Teniendo la lista anterior podemos usarlo como ejercicio para crear nuestra librería de **packetReader**.

¡Atención! acá viene la explicación sobre que significa el chorizo anterior así que atentos.

como he dicho todo es un constante intercambio de base constante, que si pasar de hexadecimal a decimal, hexadecimal a binario etc… pero hay que tener siempre claro que todo es igual, decir esto: FF(hexadecimal) es equivalente a 255(decimal) que vendría siendo equivalente a 11111111 (binario), es exactamente igual.

Ahora el cliente nos envía el mensaje anterior, y volvamos a recordar que cada sección del array es de 8bits es decir que el máximo en decimal que puede albergar cada sección del array es de 0 a 255 en decimal, entonces ahora descompongamos el paquete para poder obtener estas variables: HEADER y BODY.

Teniendo en cuenta el paquete se divide de la siguiente manera.

Los cuatro primeros bytes (32bits) es el **LENGTH,** sirve para contar los bytes (8 bits) que tiene el mensaje total o el cuerpo que se lee, es decir serían los bytes(8 bits) sumados del **header/id** + **string length** + el **body**, esto lo veremos mas a fondo en la siguiente parte de la guía (**Momentum 2**), ya cuando creemos nuestro empaquetador.

Los dos siguientes bytes (16bits) es del **HEADER** (importante para saber que evento se está haciendo en ese instante).

Los otros dos siguientes bytes (16bits) son del **STRING LENGTH** del packet (se usa para contar los caracteres del mensaje).

Y el resto de bytes son del BODY.

Nota: en la parte del handshake solo se usa este tipo de orden en los paquetes, sin embargo en el futuro se tiene que modificar para agregar Boole y un lector de enteros, como solo nos concierne el handshake lo dejaremos tal cual.

Ahora os mostraré lo explicado anteriormente usando el paquete que nos han enviado imaginariamente.

[0,0,0,21,	9,159,	0,17,	1,145,0,61,104,116,116,112,58,47,47,108,111,99,97,108,104]
Length	Header/id	StringLength	BODY

Este paquete es para uso de ejemplo, realmente los paquetes pueden ser mas largos, igualmente este sirve para hacer el packet reader, este orden se repite en todos los paquetes.

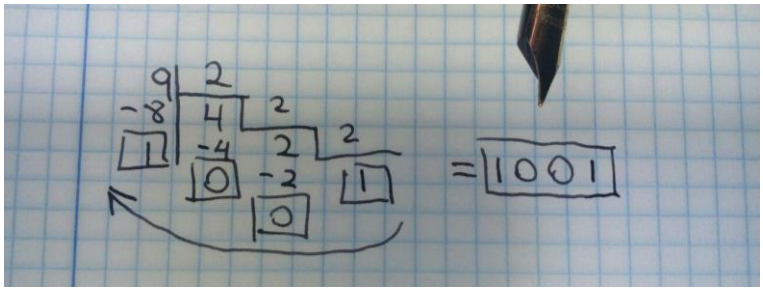
Obtención del HEADER/ID

Para obtener el valor del **HEADER** procedemos de la siguiente manera:

Hemos dicho que el valor del **HEADER** es un entero de 16bits pues tenemos que convertir el numero [9, 159] (los hemos sacado de la tabla que he puesto).

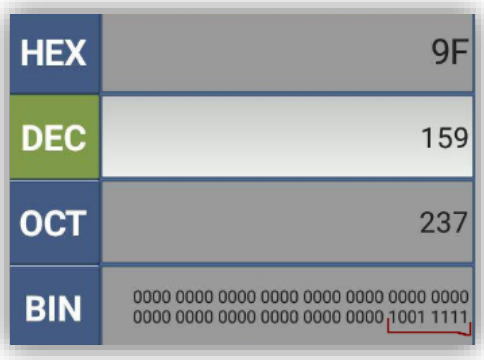
Procedamos a la conversión a **binario**, sí, he dicho a binario (todo mundo celebra al oír esta frase).

El 9 en binario ¿cómo se hace niños?, procedamos a las divisiones entre 2 sucesivas, obteniendo su resto: 9/2(resto: 1)= 4, 4/2(resto 0)=2, 2/2(resto 0)= (1), invertimos el resultado y cogemos el resto, y nos da como resultado 1001.



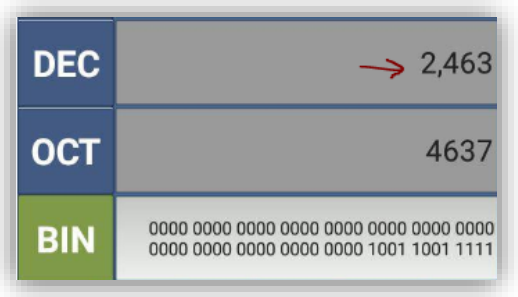
Pero tranquilos, no es necesario hacerlo todo a mano yo les recomiendo que se descarguen la aplicación para **android "DevCalc"** nos ayudará con esto de la conversión, usando la calculadora de DevCalc veremos que nos da el 9 en binario:

Nos da: 0000**1001** (si nos da menos de 8 cifras pues rellenamos con ceros la izquierda para completar las ocho cifras). Ahora vamos a calcular el valor de 159, que en binario seria:



El resultado de 159 es: 0000**1001 1111**, hacemos el cambio de nuestro array que era, [9, 159] por su valor binario **[00001001, 10011111]**, podemos observar ahora que tenemos su valor en binario y tendremos que unirlos, para poder calcular el valor de un entero de 16 bits.

Uniendo nuestro array queda así **0000100110011111** un hermoso número binario de 16 bits, y procedemos a calcular su valor en DECIMAL, y el resultado de nuestra conversión con la calculadora es de: **2463.**



Ya sabemos cuál es el valor de nuestro HEADER y realmente ese valor es un valor real que es usado para un evento y es el correcto y con esto y nuestro 2463 ya sabemos como hacer los HEADER vamos al siguiente paso: el **LENGTH.**

STRING LENGTH

Para obtener el length string se procede igual que el HEADER así que no me centraré mucho con esta conversión, el length string es útil para contar el tamaño del mensaje y así saber que parte de lo que nos llega es necesario.

[0,0,0,21,	9,159,	0,17,	1,145,0,61,104,116,116,112,58,47,47,108,111,99,97,108,104]
length	Header	lengthString	BODY

En el length string tenemos [0, 17] los convertimos a binario y obtenemos [00000000, 00010001] es decir que tenemos en total un decimal de 17, puede que sea el mismo, pero igualmente hay que hacer la conversión ya que si hay algo en la primera parte del array, su respuesta es otra, este lengh string lo guardamos ya que lo usaremos para contar los caracteres útiles del body.

BODY

Hemos dicho que el resto de bytes después de los bytes del **id**, son los bytes del **BODY** es decir que tenemos que eliminar los primeros 64 bits (o los primeros 8bytes)

[0,0,0,21, 9,159, 0,17, 1,145,0,61,104,116,116,112,58,47,47,108,111,99,97,108,104]

Después de eliminarnos nos quedaría este array:

[1,145,0,61,104,116,116,112,58,47,47,108,111,99,97,108,104]

Este array se tiene que tratar de una manera distinta al HEADER ahora nos interesa decodificar el mensaje, el mensaje viene codificado en este array en el cual cada elemento es de 8bits, entonces lo que tenemos que hacer es usar una tabla ASCII y convertir los números de cada elemento en caracteres, observemos esta tabla y los vamos haciendo poco a poco

El código ASCII – www.elCodigoASCII.com.ar

sigla en inglés de American Standard Code for Information Interchange
(Código Estadounidense Estándar para el Intercambio de Información)

Caracteres de control ASCII

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

Caracteres ASCII imprimibles

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

elCodigoASCII.com.ar

ASCII extendido

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	á	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	▀	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Hacemos la conversión de números decimales (de 8 bits) a códigos usando la tabla ascii (cada lenguaje de programación tiene su propia herramienta para pasar de decimal a caracteres alfa-numéricos).

1	145	0	61	104	116	116	112	58	47	47	108	111	99	97	108	104
inicio	ae	nul	=	h	t	t	p	:	/	/	l	o	c	a	l	h

Y bueno el mensaje final que resulta es: "inicio encabezado, ae, null, = http://localh" ¿y que pasó acá? Tenemos caracteres súper raros y curiosos, pues en este caso de ejemplo es normal, ya que el cliente nos está mandando ese mensaje tal cual, hay paquetes que hay que eliminar caracteres al final y para eso se usa el length string este nos dice hasta donde llega el mensaje, pues se implementa el length string y luego obtienes el mensaje limpio.

Ejercicio para esta guía

Por si no nos queda claro, vamos a trabajar el siguiente paquete real, el paquete pertenece al evento sso.ticket que es la autenticación del usuario, el usuario nos envía su autenticación (en este instante no nos interesa realmente lo que vamos a hacer con este paquete, solo queremos hacer nuestro readPacket).

Tenemos el siguiente paquete:

[0, 0, 0, 51, 0, 127, 0, 43, 72, 65, 66, 66, 79, 45, 52, 54, 53, 47, 53, 99, 48, 99, 51, 54, 57, 48, 57, 100, 102, 54, 50, 101, 99, 49, 99, 56, 51, 101, 52, 56, 56, 50, 100, 54, 102, 52, 99, 54, 49, 99, 98, 0, 0, 39, 27]

Separemos cada parte del paquete.

[0, 0, 0, 51, 0, 127, 0, 43, 72, 65, 66, 66, 79, 45, 52, 54, 53, 47, 53, 99, 48, 99, 51, 54, 57, 48, 57, 100, 102, 54, 50, 101, 99, 49, 99, 56, 51, 101, 52, 56, 56, 50, 100, 54, 102, 52, 99, 54, 49, 99, 98, 0, 0, 39, 27]

El color rojo es el Length, el color verde es el HEADER/ID, el color azul es el string length, el color violeta es el BODY.

Primero sacaremos el **HEADER:** [0, 127] -> convertimos a binario [00000000, 01111111] unimos esos binarios y nos queda: 0000000001111111 -> convertimos a decimal y nos queda 127.

Ahora tenemos un HEADER = 127.

Vamos a decodificar nuestro **STRING LENGTH**: [0, 43] -> convertimos a binario [00000000, 00101011] unimos esos binarios y nos queda: 0000000000101011 -> convertimos a decimal y nos queda 43.

Ahora tenemos nuestro STRING LENGTH = 43.

Vamos a decodificar nuestro **BODY:** [72, 65, 66, 66, 79, 45, 52, 54, 53, 47, 53, 99, 48, 99, 51, 54, 57, 48, 57, 100, 102, 54, 50, 101, 99, 49, 99, 56, 51, 101, 52, 56, 56, 50, 100, 54, 102, 52, 99, 54, 49, 99, 98, 0, 0, 39, 27]

Como el BODY es muy largo, lo separaré de diez en diez y lo decodificaré.

72	65	66	66	79	45	52	54	53	47
H	A	B	B	O	-	4	6	5	/

53	99	48	99	51	54	57	48	57	100
5	c	0	c	3	6	9	0	9	d

102	54	50	101	99	49	99	56	51	101
f	6	2	e	c	1	c	8	3	e

52	56	56	50	100	54	102	52	99	54
4	8	8	2	d	6	f	4	c	6

49	99	98	0	39	27
1	c	b	NULL	‘	ESC

Si nos fijamos en el mensaje la respuesta es:

HABBO-465/5c0c36909df62ec1c83e4882d6f4c61cb[NULL][‘][ESC]

Podemos observar que al final de nuestro mensaje hay caracteres que no deberían de ir ahí, pues usaremos nuestro **STRING LENGTH** para contar los caracteres válidos, recordemos que nuestro String Length es de 43, procedemos a contar

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
H	A	B	B	O	-	4	6	5	/	5	c	0	c	3	6	9	0	9	D	F	6	2

24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	--	--	--
e	c	1	c	8	3	e	4	8	8	2	d	6	f	4	c	6	1	c	b	Nu	‘	es

Hemos observado que haciendo la comparación con el String Length nos queda un mensaje limpio.

En conclusión, necesitaremos crear una función/clase que nos haga todo lo que hemos comentado anteriormente, esta función/clase se le tienen que pasar los paquetes completos y esta función/clase devolverá todo lo antes mencionado (gracias a sus métodos), la función/clase deberá tener los siguientes métodos:

1. **Lector de header/id**
2. **Lector de body**

3. **Lector de String Lenght**

Nota: más adelante después del handshake se tienen que agregar mas métodos como pueden ser el lector de Boole y el lector de enteros.

Con todo esto y nuestra función, podremos crear cada parte del paquete y obtener su resultado, os mostraré el resultado que da mi función hecha en NodeJS:

```
λ node test

HEADER: 127
Leyendo string: "HABBO-465/5c0c36909df62ec1c83e4882d6f4c61cb"
```

Ahora agregamos nuestra función a nuestra prueba de emulador e intentamos hacer el handshake una vez mas, iniciamos el emulador y cargamos el cliente:

```
C:\Users\gammafp\Desktop\EMUS\gamma  (gamma@1.0.0)
λ node app

server iniciado en el puerto 3030
HEADER: 26979
BODY: "file-request/>\u0000"
--------------------------------


Se ha salido el usuario:
```

Por fin, ya tenemos las cabeceras, ahora vosotros preguntaréis ¿y donde obtenemos los HEADERS y que significan cada número? R: cada numero del header se obtiene del cliente, la comunidad es muy activa al respecto y publica los headers actualizado para cada versión, yo solo pondré los headers del handshake de esta versión.

Continuemos, ahora vemos que el header que nos llega es el 26979, pues por ahora nos bastará con poner un if y enviamos el policy, reniciamos el servidor y el cliente y observamos que nos pide ahora.

```
const server = net.createServer((socket) => {

    socket.on('data', (data) => {
            // Creación de Paquetes
            var datos = new Packet(data);
            var HEADER = datos.header;
            var body = datos.readString();

            log.data("--------------------------------");
            log.data("HEADER: "+ HEADER);
            log.data("BODY: "+ body);

            // Evento del policy
            if(HEADER === 26979) {
                log.info("------------------------");
                log.info('Se ha enviado el policy');
                socket.write(policy);
            }


    });
```

Ahora nos fijamos que nos dice el emulador:

```
C:\Users\gammafp\Desktop\EMUS\gamma  (gamma@1.0.0)
λ node app

server iniciado en el puerto 3030

--------------------------------

HEADER: 26979

BODY: file-request/>

------------------------

Se ha enviado el policy

Se ha salido el usuario:

Se ha salido el usuario:

--------------------------------

HEADER: 4000

BODY: PRODUCTION-201607262204-86871104

--------------------------------

HEADER: 340

BODY:
```

Interesante, ahora nos manda un header 4000 eso solo significa que nos ha mandado su versión, y vemos que también nos manda un **HEADER 340** este sí que es importante, este significa que nos pide iniciar la encriptación y viene la parte más divertida, pero eso lo dejaremos en el siguiente capítulo…

FIN DE LA PRIMERA PARTE.

En el siguiente capítulo veremos sobre el creador de paquetes para enviarlos al cliente un poco sobre DiffieHellman y RSA.

Recordad que todo esto ha costado mucho deducir, he preguntado a muchas personas sobre todo al que descubrió como funciona todo esto, en la parte de la estructura del paquete me comentó que era distinto y tiene toda la razón, sin embargo en la parte del handshake no se llega a usar la estructura que él me comentó y vi mas razonable poner esa estructura solo para la introducción de todo este mundillo.

Os pido que si seguís esta guía la uséis con el afán de aprender más y sé que yo también me equivoco y a lo mejor tengo muchas cosas mal, solo os pido que si tengo algo mal me agreguéis a Skype con el usuario **gammafp** y me comentéis que es lo que merece cambiar (no hablo del diseño del pdf, sino de la teoría del emulador).

La guía Momentum es solo para aprendizaje y es gratuita.

TAREAS PARA AVANZAR EN EL SIGUIENTE CAPITULO:

Leer este artículo sobre DIFFIE HELLMAN: CLICAME, CLICA_VIDEO

Leer este artículo sobre RSA: CLICAME, CLICA_VIDEO

Leer este artículo sobre RC4: CLICAME, CLICA_VIDEO