Ghidra script to decrypt a string array in XOR DDoS

maxkersten.nl/binary-analysis-course/analysis-scripts/ghidra-script-to-decrypt-a-string-array-in-xor-ddos/

This article was published on the 25th of July 2021. This article was updated on the 8th of December 2021.

The XOR DDoS bot, an ELF file for Linux distributions, is used to perform DDoS attacks. This article focuses on a rather small segment of the malware family: the internally used encrypted string array, and its decryption. This article will dive into arrays, the decryption loop, the decryption routine, and the creation of a Ghidra script in Java to automate this process.

Table of contents

The sample

The sample can be downloaded from VirusBay, Malware Bazaar, or MalShare. The hashes are given below.

```
MD5: 349456ecaa1380a142f15810a8260378
SHA-1: 02dd15ecdeedefd7a2f82ba0df38703a74489af3
SHA-256: 0f00c2e074c6284c556040012ef23357853ccac4ad1373d1dea683562dc24bca
Size: 625889 bytes
```

Used tooling

The analysis in this article has been done with a self-built version of Ghidra. The used sources date back to the first of June 2021. The used for-loop that is seen later in this article is displayed as a while-loop in earlier Ghidra versions. Other than that, no significant changes are present between versions. All analysis options have been used when analysing the file.

Arrays in theory

Arrays, disregarding of the type, are structured the same way. The first element, which resides at index zero, marks the start of the array, followed by the other elements if present. To obtain the element for a given index, the size of each element is multiplied by the index number, which is then added to the address of the first element. The code below provides an example in pseudo code, where *T* is the element's type.

```
long elementAddress = arrayBase + (sizeof(T) * i);
```

When dealing with strings, the length can be variable, as not all strings have the same length. In some cases, strings with a fixed length are used, this uses more memory than is required, but makes it easy to find the element for a given index. Alternatively, the string's length can be calculated using strlen, which is then used instead of the sizeof function. This requires more CPU cycles, as the string length needs to be calculated, but it uses less memory.

Understanding the loop

The given sample contains symbols, which make the analysis easier. The main function is already called *main*, and contains the decryption loop for the string array at *0804d12a*. The complete loop's assembly code is given below.

```
        MOV        dword ptr [EBP + local_3c],0x0
        JMP        LAB_0804d12e
LAB_0804d108
        MOV        EDX,dword ptr [EBP + local_3c]
        MOV        EAX,EDX
        SHL        EAX,0x2
        ADD        EAX,EDX
        SHL        EAX,0x2
        ADD        EAX,daemonname
        MOV        dword ptr [ESP + local_3dec],0x14
        MOV        dword ptr [ESP]=>local_3df0,EAX
        CALL       encrypt_code
        ADD        dword ptr [EBP + local_3c],0x1
LAB_0804d12e
        CMP        dword ptr [EBP + local_3c],0x16
        JBE        LAB_0804d108
```

The loop, like most loops in assembly languages, starts with the initialisation of a variable to store the count in. This variable is commonly named *i* when named by programmers. In this case, the variable is named *local_3c* by Ghidra. The naming scheme of Ghidra is based on the location. A jump is then made downwards, where the value of *local_3c* is compared to *0x16*, or *22* in decimal. If the value of *local_3c* is below or equal, the jump upwards is taken. The two labels, *LAB_0804d108* and *LAB_0804d12e*, can be renamed to *loop_body* and *loop_compare* respectively to increase the readability. The variable *local_3c* can be renamed to *i* for further clarification.

To understand the loop's body, each instruction will be explained below, in the usual step-by-step manner. At first, the value of *i*, which resides at *EBP* + *i* is moved into *EDX*, after which it is also moved into *EAX*.

```
MOV        EDX,dword ptr [EBP + i]
MOV        EAX,EDX
```

Next, the value in *EAX* is shifted left by two bits. A left shift of *N* equals two to the power of *N*. In this case, it means that *EAX* is multiplied by four.

```
SHL        EAX,0x2
```

The value of *i* is then added to *EAX*, after which it is multiplied by four again.

```
ADD        EAX,EDX
SHL        EAX,0x2
```

The variable named *daemonname* refers to the string array, although Ghidra does not recognise the type due to the fact that it contains the encrypted strings. The base address of the array is then added to *EAX*.

```
ADD        EAX,daemonname
```

In short, *EAX* contains the array's base address and the offset based on *i*.

The next three instructions push two arguments on the stack, after which the decryption function (named *encrypt_code*) is called. The first argument, as they are read from the stack in the reverse order, is equal to *EAX*, which contains the current element's address. The second argument is equal to *0x14*, or *20* in decimal. As such, it becomes apparent that this string array is based on strings with a fixed length, although this does not mean that every string is 20 bytes in size. Rather, the length of each string is between 0 and 19, given that each string is terminated with a null byte.

```
MOV        dword ptr [ESP + local_3dec],0x14
MOV        dword ptr [ESP]=>local_3df0,EAX
CALL       encrypt_code
```

At last, the value of *i* is incremented with one.

```
ADD        dword ptr [EBP + i],0x1
```

The calculation for the offset of the next element can be simplified as follows.

```
for (int i = 0; i <= 0x16; i++)
{
        int result = i;
        result = result * 4;
        result += i;
        result = result * 4;
        System.out.println(result);
}
```

Further simplified, one can rewrite the code above as follows.

```
for (int i = 0; i <= 0x16; i++)
{
        int result = ((i * 4) + i) * 4;
        System.out.println(result);
}
```

Since multiplication is the same as repeated addition, one can even further simplify the formula. At first, *i* is multiplied by four, after which *i* is added. This can be simplified by stating that *i* is multiplied by five. The outcome of this is multiplied by four. As such, the original value is multiplied by four, after which its multiplied by five. In total, *i* is multiplied by (four times five) twenty. The code below shows the simplification in several steps.

```
((i * 4) + i) * 4;
(i * 5) * 4
i * 20
```

The likely reason as to why the code looks like this, is the efficiency of the shift instructions, when compared to the multiplication instructions. The compiler likely chose to replace a single multiplication with less resource intensive instructions.

When looking at the *main* function in Ghidra's decompiler, one can find the string array decryption loop at line 100, 101, and 102. The excerpt is given below. Note that the refactoring of the variables in the assembly code is reflected in this code.

```
for (i = 0; i < 0x17; i = i + 1) {
  encrypt_code(daemonname + i * 0x14,0x14);
}
```

Note that this loop displays *0x17* with regards to the amount of iterations, rather than *0x16*. The condition for the loop is *less than*, rather than *less than or equal*, meaning the value needs to be incremented with one.

Fully understanding the code, and how it is generated, will be useful when creating a Ghidra script later on.

Remaking the decryption routine

The decryption function, named *encrypt_code*, is used to decrypt a given encrypted string with a given length. The function is given below.

```
byte * encrypt_code(byte *param_1,int param_2)
{
  byte *local_10;
  int local_c;

  local_10 = param_1;
  for (local_c = 0; local_c < param_2; local_c = local_c + 1) {
    *local_10 = *local_10 ^ xorkeys[local_c % 0x10];
    local_10 = local_10 + 1;
  }
  return param_1;
}
```

The first argument (named *param_1*) can be renamed into *input*, whereas the second argument (named *param_2*) can be renamed into *length*. The variable named *local_10* is a copy of the given input, as it points to the same value. As such, it can be renamed into *inputCopy*. The loop uses *local_c* as its counter, which can be renamed into *i*. The refactored code is given below.

```
byte * encrypt_code(byte *input,int length)
{
  byte *inputCopy;
  int i;

  inputCopy = input;
  for (i = 0; i < length; i = i + 1) {
    *inputCopy = *inputCopy ^ xorkeys[i % 0x10];
    inputCopy = inputCopy + 1;
  }
  return input;
}
```

The variable *xorkeys* is a string, although Ghidra does not recognise it as such. Changing the type, using *T* as a hotkey in the disassembly view, will display its content. Alternatively, one can also get the raw value of the bytes instead. The key equals *BB2FA36AAA9541F0*.

When rewriting the decryption function in Java, there is one more more thing to take into account. Strings in C end with a null byte, but when using a byte array to create a string in Java, this byte is to be omitted. As not all strings are equal to the predefined length, a check is to be included to break the loop when the null byte is encountered. When breaking the loop, the bytes that have been decrypted thus far are to be used to create a new string. The recreated function is given below.

```
private String decrypt(byte[] input, char[] key) {
  byte[] output = new byte[input.length];

  for (int i = 0; i < input.length; i++) {
    if(input[i] == 0) {
      break;
    }
    output[i] = (byte) (input[i] ^ key[i % 0x10]);
  }
  return new String(output);
}
```

Note that the key is passed as an argument to the function, as this will come in useful when creating the script. By passing the value as an argument to the function, one can keep all variables in a single place within the script.

Writing the Ghidra script

The script itself uses the decryption function that was created in the previous step. To decrypt the string array, several variables need to be initialised first. The decryption *key*, as is required by the decryption function, as well as the location of the array (defined as *arrayBase*), the size of a single element (defined as *elementSize*), and the amount of elements of the array (defined as *arraySize*).

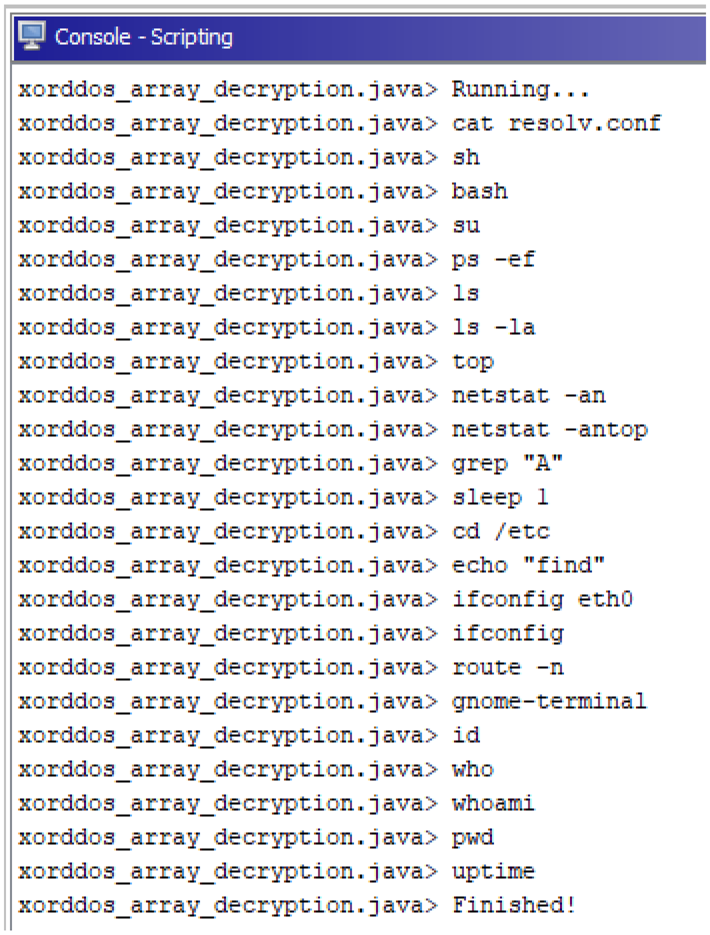
```
char[] key = "BB2FA36AAA9541F0".toCharArray();
int arrayBase = 0x080cf1c0;
int elementSize = 0x14;
int arraySize = 0x17;
```

To get the value of each element, one needs to multiply the loop count with the predefined element size, after which the array's base address is added. To get the data from the sample, one can use the getBytes function, which requires an Address to know where to start reading the bytes from, and an integer to know how many bytes should be read. To convert an integer, long, or string to an Address object, one needs to use the toAddr function.

The obtained bytes are decrypted by the decryption function, along with the decryption key. The result is then printed to Ghidra's console.

```
try {
  for (int i = 0; i < arraySize; i++) {
        int offset = i * elementSize;
        int location = arrayBase + offset;
        byte[] input = getBytes(toAddr(location), elementSize);
        String decrypted = decrypt(input, key);
        println(decrypted);
  }
} catch (MemoryAccessException e) {
  e.printStackTrace();
  println("\nA memory access exception occurred, please refer to the stacktrace above
for more information");
}
```

In the case of an error with the *getBytes* function, a *MemoryAccessException* is thrown. The image below shows the output of the script once its execution has finished.



Console - Scripting

```
xorddos_array_decryption.java> Running...
xorddos_array_decryption.java> cat resolv.conf
xorddos_array_decryption.java> sh
xorddos_array_decryption.java> bash
xorddos_array_decryption.java> su
xorddos_array_decryption.java> ps -ef
xorddos_array_decryption.java> ls
xorddos_array_decryption.java> ls -la
xorddos_array_decryption.java> top
xorddos_array_decryption.java> netstat -an
xorddos_array_decryption.java> netstat -antop
xorddos_array_decryption.java> grep "A"
xorddos_array_decryption.java> sleep 1
xorddos_array_decryption.java> cd /etc
xorddos_array_decryption.java> echo "find"
xorddos_array_decryption.java> ifconfig eth0
xorddos_array_decryption.java> ifconfig
xorddos_array_decryption.java> route -n
xorddos_array_decryption.java> gnome-terminal
xorddos_array_decryption.java> id
xorddos_array_decryption.java> who
xorddos_array_decryption.java> whoami
xorddos_array_decryption.java> pwd
xorddos_array_decryption.java> uptime
xorddos_array_decryption.java> Finished!
```

Conclusion

Decrypting content from a sample provides a lot more insight as to what the sample does, especially because these strings are concealed for a reason. In some cases, bots within the same family reuse the encryption key. If the key changes, it is easy to replace it in the script, or make use of a dialog in the script that requests the key once executed.

Understanding how to easily access variables and memory in a script in Ghidra is helpful when analysing any sample. Given that some code segments are easy to reuse, it is useful to create scripts that are made up of easily reusable functions.

The complete script

The complete script, including documentation, is given below. Note the hardcoded key, element length, and array length. For more information on how to avoid hardcoding values in a script, one can visit the Amadey string decryption script article.

```
//This script is used to decrypt a string array within the XOR DDoS bot. Note that it
the array's location, element size, and array size are hardcoded in the script.
//@author Max 'Libra' Kersten (https://maxkersten.nl, @Libranalysis)
//@category string array decryption
//@keybinding
//@menupath
//@toolbar

import ghidra.app.script.GhidraScript;
import ghidra.program.model.mem.MemoryAccessException;

public class xorddos_array_decryption extends GhidraScript {

  /**
   * This function is called by Ghidra, as such it is the entry into the script
   */
  @Override
  protected void run() throws Exception {
    //The hardcoded decryption key, works for numerous samples, but this script is
based on 0f00c2e074c6284c556040012ef23357853ccac4ad1373d1dea683562dc24bca
    char[] key = "BB2FA36AAA9541F0".toCharArray();
    //The location of the array's base address, which resides at index 0
    int arrayBase = 0x080cf1c0;
    //The size of each element within the array
    int elementSize = 0x14;
    //The amount of elements in the array
    int arraySize = 0x17;
    try {
      //Since the assembly code uses a Jump Below or Equal instruction to compare i
to 0x16, the loop needs to iterate 0x17 times to cover the complete array
      for (int i = 0; i < arraySize; i++) {
        //Declare the offset from the array base (meaning the size of each element
times the element that is selected)
        int offset = i * elementSize;
        //Declare the location within the program
        int location = arrayBase + offset;
        //Get the bytes for one element at the given location
        byte[] input = getBytes(toAddr(location), elementSize);
        //Decrypt the given data using the given key
        String decrypted = decrypt(input, key);
        //Print the decrypted string
        println(decrypted);
      }
    } catch (MemoryAccessException e) {
      //Print the stacktrace, and provide further indication that an error occured
      e.printStackTrace();
      println("\nA memory access exception occurred, please refer to the stacktrace
above for more information");
    }
  }

  /**
   * Decrypts the given input using the given key and returns a new string with the
decrypted content in it
   * @param input the data to decrypt
   * @param key the key to decrypt the given input
   * @return the decrypted input in the form of a string
   */
  private String decrypt(byte[] input, char[] key) {
    //Create a new byte array to store the decrypted data in
    byte[] output = new byte[input.length];
    //Iterate over all bytes
    for (int i = 0; i < input.length; i++) {
      //If the byte equals zero, the string is terminated, meaning the loop needs to
be broken, which returns a string based on the data that has been decrypted thus far
      if(input[i] == 0) {
        break;
      }
      //Decrypt the current byte
      output[i] = (byte) (input[i] ^ key[i % 0x10]);
    }
    //Once the loop breaks, or all iterations have finished, a new string is returned
    return new String(output);
  }
}
```