Ghidra script to handle stack strings

maxkersten.nl/binary-analysis-course/analysis-scripts/ghidra-script-to-handle-stack-strings/

This article was published on the 12th of April 2022.

The usage of stack strings within malware is frequent, especially when analysing shellcode or more advanced samples. This article will provide insight into stack strings, the endianness of values, how to handle (encrypted) stack strings, and the step-by-step creation of a Ghidra script which handles (wide) stack strings, which may or may not be encrypted. At last, the complete script is given.

The analysis in this article is done with a self-built version of Ghidra 10.2, based on the publicly available source code of the first of February 2022. The samples have been analysed using all analysers.

Table of contents

Stack strings explained

In general, strings are not stored within the text segment of the binary. When a string is used in a local variable in source code, the compiler generally places the string in a different segment. A pointer to the string’s location is used to access or alter the data.

To obfuscate strings, one can create a string on the stack, character for character, or in small groups of characters. The concatenation of these characters lead to the existence of the complete string on the stack. When viewing the strings of a binary, the characters might not show up.

As an example, GNU strings only shows human readable strings with a length of 4 or longer by default, meaning that smaller groups do not show up, or might show up in parts. The third crack me in this course is based on this technique. Note that Mandiant’s FLOSS also aims to detect stack strings in binaries.

Additionally, stack strings can also be encrypted. This would make the use of tools harder, although FLOSS attempts to also decrypt stack strings if a simple encryption scheme is used.

Note that the character set and size of the used encoding matters. To make the script more generic, the obtained stack string bytes will be printed out using multiple encodings. The wrong encoding is likely to be displayed as garbage, making it fairly easy for an analyst to pick the string that looks correct.

Endianness

The endianness defines where the most significant bit is, which is either the first or the last. The basic CPU architecture article explains the endianness in more detail. With stack strings, it is important to understand if several characters need to be read right-to-left, or left-to-right.

Stack string example: CaddyWiper

CaddyWiper is a wiper which was used by a pro-Russian actor against Ukrainian victims in March 2022, after the Russian invasion, as reported by ESET. It contains a lot of stack strings, including wide strings. The hashes of the sample are given below. The sample can be downloaded from Malware Bazaar and MalShare.

```
MD-5: 42e52b8daf63e6e26c3aa91e7e971492

SHA-1: 98b3fb74b3e8b3f9b05a82473551c5a77b576d54

SHA-256: a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea
```

The excerpt below shows a wide stack string, which can be seen due to the frequent occurrence of *0x00*, and the double zero to terminate the string.

```
MOV  byte ptr [EBP + local_20],0x6b
MOV  byte ptr [EBP + local_1f],0x0
MOV  byte ptr [EBP + local_1e],0x65
MOV  byte ptr [EBP + local_1d],0x0
MOV  byte ptr [EBP + local_1c],0x72
MOV  byte ptr [EBP + local_1b],0x0
MOV  byte ptr [EBP + local_1a],0x6e
MOV  byte ptr [EBP + local_19],0x0
MOV  byte ptr [EBP + local_18],0x65
MOV  byte ptr [EBP + local_17],0x0
MOV  byte ptr [EBP + local_16],0x6c
MOV  byte ptr [EBP + local_15],0x0
MOV  byte ptr [EBP + local_14],0x33
MOV  byte ptr [EBP + local_13],0x0
MOV  byte ptr [EBP + local_12],0x32
MOV  byte ptr [EBP + local_11],0x0
MOV  byte ptr [EBP + local_10],0x2e
MOV  byte ptr [EBP + local_f],0x0
MOV  byte ptr [EBP + local_e],0x64
MOV  byte ptr [EBP + local_d],0x0
MOV  byte ptr [EBP + local_c],0x6c
MOV  byte ptr [EBP + local_b],0x0
MOV  byte ptr [EBP + local_a],0x6c
MOV  byte ptr [EBP + local_9],0x0
MOV  byte ptr [EBP + local_8],0x0
MOV  byte ptr [EBP + local_7],0x0
```

If one were to rename and retype the first variable, originally named *local_20*, to a 26 character long array named *s_kernel32.dll*, the string's memory lay-out becomes apparent, as can be seen below.

```
MOV  byte ptr [EBP + s_kernel32.dll[0]],0x6b
MOV  byte ptr [EBP + s_kernel32.dll[1]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[2]],0x65
MOV  byte ptr [EBP + s_kernel32.dll[3]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[4]],0x72
MOV  byte ptr [EBP + s_kernel32.dll[5]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[6]],0x6e
MOV  byte ptr [EBP + s_kernel32.dll[7]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[8]],0x65
MOV  byte ptr [EBP + s_kernel32.dll[9]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[10]],0x6c
MOV  byte ptr [EBP + s_kernel32.dll[11]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[12]],0x33
MOV  byte ptr [EBP + s_kernel32.dll[13]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[14]],0x32
MOV  byte ptr [EBP + s_kernel32.dll[15]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[16]],0x2e
MOV  byte ptr [EBP + s_kernel32.dll[17]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[18]],0x64
MOV  byte ptr [EBP + s_kernel32.dll[19]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[20]],0x6c
MOV  byte ptr [EBP + s_kernel32.dll[21]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[22]],0x6c
MOV  byte ptr [EBP + s_kernel32.dll[23]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[24]],0x0
MOV  byte ptr [EBP + s_kernel32.dll[25]],0x0
```

The generic stack string handling suffices to deal with these kind of (wide) stack strings.

Stack string example: PlugX

Talisman is a variant of PlugX, which is a remote access trojan which is generally used by Chinese related actors. This backdoor contains encrypted stack strings, of which the hashes are given below. The samples can be downloaded from Malware Bazaar and MalShare. Note that the referenced sample is dumped from memory, as it is initially encrypted. Details about the Talisman variant can be found in a corporate blog I co-authored.

```
MD-5: c6c6162cca729c4da879879b126d27c0
```

```
SHA-1: 80e5fd86127de526be75ef42ebc390fb0d559791
```

```
SHA-256: 344fc6c3211e169593ab1345a5cfa9bcb46a4604fe61ab212c9316c0d72b0865
```

One of the encrypted stack strings within the sample can be found at the offset of *0x100002401*. The assembly instructions at the given offset is given below.

```
MOV  dword ptr [ESP + local_cc],0x55615596
MOV  dword ptr [ESP + local_c8],0x5573555e
MOV  dword ptr [ESP + local_c4],0x55615574
MOV  dword ptr [ESP + local_c0],0x55995571
MOV  dword ptr [ESP + local_bc],0x55615578
MOV  dword ptr [ESP + local_b8],0x55785582
MOV  dword ptr [ESP + local_b4],0x55775561
MOV  dword ptr [ESP + local_b0],0x5538553d
MOV  dword ptr [ESP + local_ac],0x551f552d
MOV  dword ptr [ESP + local_a8],0x558d552d
MOV  dword ptr [ESP + local_a4],0x5555553c
MOV  word ptr [ESP + local_a0],0x5555
```

The local variable named *local_cc* is the start of the stack string, where all of the encrypted characters are placed on the stack in order. The code below, as seen in the decompiled code, decrypts the given stack string.

```
do {
  bVar4 = (*(char *)((int)&local_cc + uVar2) + 0x22U ^ 0x33) + 0xbc;
  param_1 = param_1 & 0xffffff00 | (uint)bVar4;
  *(byte *)((int)&local_cc + uVar2) = bVar4;
  uVar2 = uVar2 + 1;
} while (uVar2 < 0x2e);
```

The decompiled code is a bit cluttered, but can easily be cleaned up. The variable named *uVar2* is the loop's counter, which can be renamed to *i*. The line with the reference to *param_1* can be omitted, and the local variable *bVar4* can be renamed to *decryptedByte*. Upon doing so, the code is becomes much more readable, as can be seen below.

```
do {
  decryptedByte = (*(char *)((int)&local_cc + i) + 0x22U ^ 0x33) + 0xbc;
  *(byte *)((int)&local_cc + i) = decryptedByte;
  i = i + 1;
} while (i < 0x2e);
```

There is, however, a mistake in the decompiled code. The decryption routine seems straight forward: add *0x22*, xor with *0x33*, and add *0xbc*. However, when looking at the assembly instructions, one can easily spot the difference, as can be seen in the code below.

```
ADD  param_1,0x22
XOR  param_1,0x33
SUB  param_1,0x44
```

The used variable names in the decompiler differ somewhat from the disassembly view, but the *param_1* variable in the code above, is the current byte in the loop. The last instruction subtracts *0x44*, rather than adding *0xbc*. Knowing this, the decryption routine can be re-made, as can be seen below.

```
/*
 * Decrypts the string based on the algorithm within the sample (SHA-256
 * 344fc6c3211e169593ab1345a5cfa9bcb46a4604fe61ab212c9316c0d72b0865, offset
 * 0x10002460)
 */
private byte[] decrypt(List<Integer> input) {
    // Initialise the output variable
    byte[] output = new byte[input.size()];
    // Loop over the input and decrypt the given byte
    for (int i = 0; i < input.size(); i++) {
        output[i] = (byte) (((input.get(i) + 0x22) ^ 0x33) - 0x44);
    }
    // Return the decrypted output
    return output;
}
```

Writing the Ghidra script

The script is based on Mich's Python based SimpleStackStrings Ghidra script. The script in this article is written in Java, as this is Ghidra's native way of scripting.

This script will use the currently selected address in Ghidra as a variable, rather than requesting input from the user to provide the address via one of the ask-related functions. The code for which is given below, where *currentAddress* is inherited from the GhidraScript class.

Next, the script needs to iterate over all instructions that are part of the stack string. The *Instruction* object has a function which returns the instruction that is directly below it, which is *getNext*. This function returns *null* if there is no instruction. As such, a while-loop with the condition that the instruction is not *null*. The skeleton code is given below.

```
while(instruction != null) {
    //TODO: implement the logic

    instruction = instruction.getNext();
}
```

Instructions can contain a value, known as a scalar value. The value can be in the first or second part of the instruction. The code below provides two examples.

```
MOV dword ptr [ESP + local_var], 0x41414141
```

```
PUSH 0x41414141
```

In the first example, the value of *0x41414141* is the second scalar value, which resides at index *1*. In the second example, there is no second scalar value, meaning the value resides at index *0*.

As such, one can attempt to obtain the second scalar value, using the *getScalar(int index)* function within the *Instruction* class, of the current instruction. If there is no such value, *null* is returned. As such, one can always attempt to fetch the second scalar value.

If the returned value is *null*, an attempt needs to be made to fetch the first scalar value. If this also returns *null*, the instruction is not part of the stack string, at least not in a format that this script supports. The code below resembles the logic that is described above.

```
Scalar scalar = instruction.getScalar(1);
    if (scalar == null) {
        scalar = instruction.getScalar(0);
        if (scalar == null) {
            println("Stack string ended, since no suitable scalar value could be
found at 0x"
                            + Long.toHexString(instruction.getAddress().getOffset())
+ "!");
            break;
         }
    }
```

Following on that, the scalar’s value needs to be obtained in hexadecimal format. The value then needs to be converted due to the endianness. The code is given below, where the *convert* function will be described afterwards.

```
long value = scalar.getValue();
String valueString = Long.toHexString(value);
valueBytes.addAll(convert(valueString));
```

The *convert* function, which is given below, takes a *String* as argument, and returns a list of boxed integers. Due to the endianness, the bytes need to be read from right-to-left. A byte is two characters from the string in size. To clarify, the value *0x41414141* consists of four bytes: *0x41 0x41 0x41 0x41*. In this case, the string does not contain the leading *0x*. To split the string up in bytes that are equal in value to their character counterpart (the value they portray, not their *char* value).

```
private List<Integer> convert(String input) {
    List<Integer> scalarValue = new ArrayList<>();
    if (input == null || input.isEmpty()) {
        return null;
    }

    int remainder = input.length() % 2;

    if (remainder != 0) {
        input = "0" + input;
    }

    for (int i = input.length(); i > 0; i = i - 2) {
        int j = i - 2;
        if (i == 0 || j < 0) {
            break;
        }
        String b = input.substring(j, i);
        Integer conversion = Integer.parseInt(b, 16);
        scalarValue.add(conversion);
    }
    return scalarValue;
}
```

If the amount of characters is not even, a leading zero needs to be added to the byte which is missing it, which is the least significant byte. The code responsible to do so is the within the if-body where the value of the *remainder* is checked.

The decryption function takes a list of integers as input, and returns an array of bytes. The code is given below.

```
private byte[] decrypt(List<Integer> input) {
    byte[] output = new byte[input.size()];
    for (int i = 0; i < input.size(); i++) {
        output[i] = (byte) (((input.get(i) + 0x22) ^ 0x33) - 0x44);
    }
    return output;
}
```

In the generic script, which does not handle string decryption, the *decrypt* function signature is the same, but the body is not. Instead, it simply converts the given list of integers into an unboxed byte array. This leaves the decryption function as a template for future use-cases in the generic script. The code is given below.

```
private byte[] decrypt(List<Integer> input) {
    byte[] output = new byte[input.size()];
    for (int i = 0; i < input.size(); i++) {
        output[i] = input.get(i).byteValue();
    }
    return output;
}
```

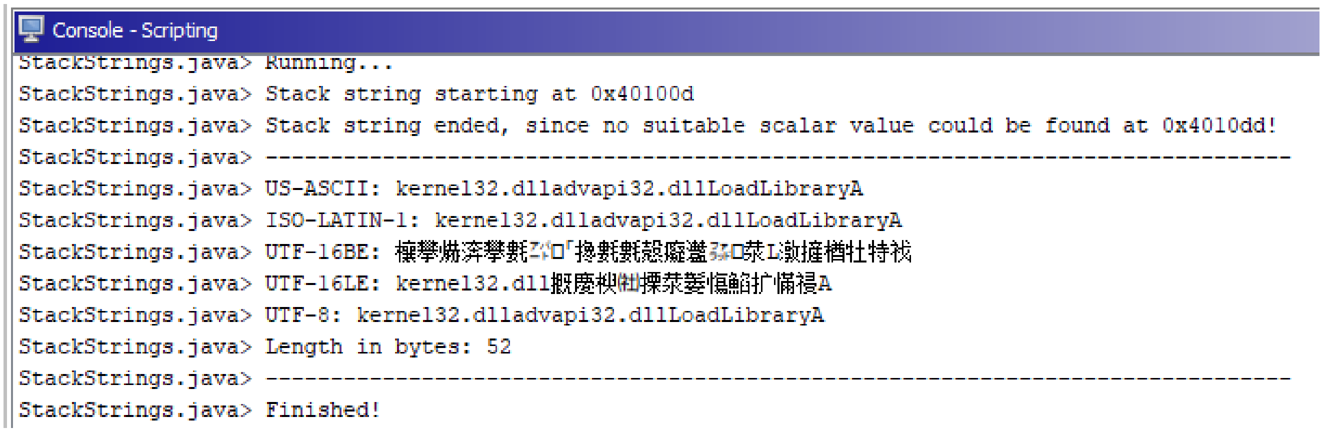
At last, the returned unboxed byte array is converted into multiple strings, all of which are based on a different character set. Since the character set is unknown to the script itself, numerous types are included. The analyst can then easily see which of the strings are garbage, and which is the correct string. The strings are then printed, together with some identifying information with regards to the character set. Additionally, the amount of bytes of the stack string is printed. This helps when changing the data type into an array of (wide) characters, where one knows the number of characters that is required for the string.

```
String usAscii = new String(output, StandardCharsets.US_ASCII);

String isoLatin1 = new String(output, StandardCharsets.ISO_8859_1);

String utf16be = new String(output, StandardCharsets.UTF_16BE);

String utf16le = new String(output, StandardCharsets.UTF_16LE);

String utf8 = new String(output, StandardCharsets.UTF_8);

println("----------------------------------------------------------------------------
---");
println("US-ASCII: " + usAscii);
println("ISO-LATIN-1: " + isoLatin1);
println("UTF-16BE: " + utf16be);
println("UTF-16LE: " + utf16le);
println("UTF-8: " + utf8);
println("Length in bytes: " + output.length);
println("----------------------------------------------------------------------------
---");
```

Do note that if the stack string consists of multiple strings before an invalid instruction is hit, all of them are shown at once. The analyst can see which string is what size, and rename and retype the variables accordingly. The following screenshots include such a case.

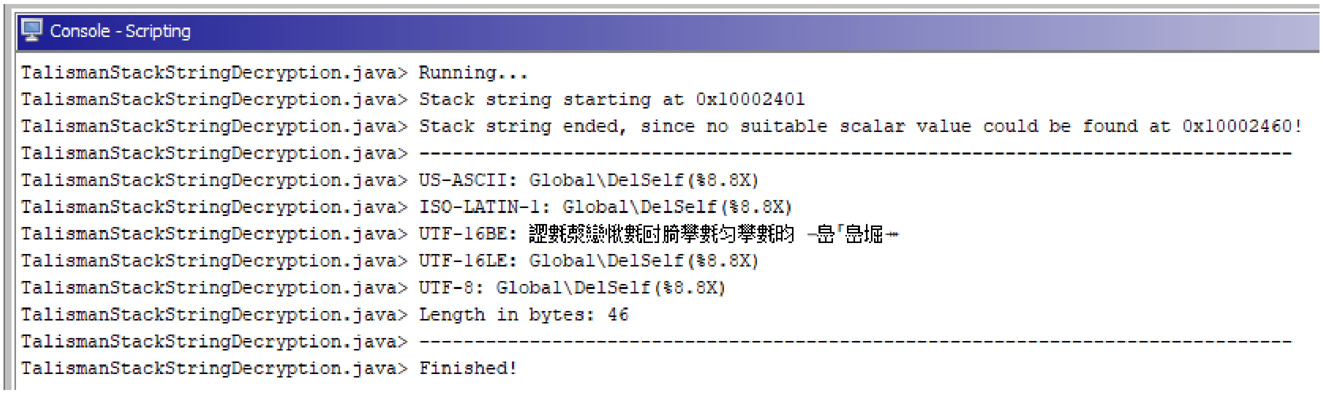
When viewing this in Ghidra’s console, it is apparent that some spaces that should be there, are not properly shown in the console. Note that this might not be the case in future Ghidra versions.

The images below show the output of Ghidra’s console, for one stack string from both CaddyWiper and PlugX Talisman respectively



```
StackStrings.java> Running...
StackStrings.java> Stack string starting at 0x40100d
StackStrings.java> Stack string ended, since no suitable scalar value could be found at 0x4010dd!
StackStrings.java> ------------------------------------------------------------------------------
StackStrings.java> US-ASCII: kernel32.dlladvapi32.dllLoadLibraryA
StackStrings.java> ISO-LATIN-1: kernel32.dlladvapi32.dllLoadLibraryA
StackStrings.java> UTF-16BE: 樺攀爀渀攀氀㌀㈀⸀搀氀氀愀搀瘀愀瀀椀㌀㈀⸀搀氀氀䰀漀愀搀䰀椀戀爀愀爀礀
StackStrings.java> UTF-16LE: kernel32.dll摡慶楰㈳搮汬潌摡楌牢牡她A
StackStrings.java> UTF-8: kernel32.dlladvapi32.dllLoadLibraryA
StackStrings.java> Length in bytes: 52
StackStrings.java> ------------------------------------------------------------------------------
StackStrings.java> Finished!
```

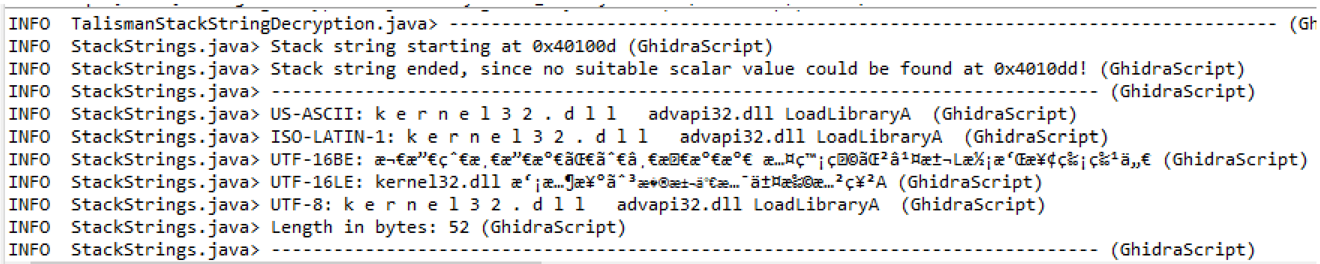
The script’s output in Ghidra’s console



```
TalismanStackStringDecryption.java> Running...
TalismanStackStringDecryption.java> Stack string starting at 0x10002401
TalismanStackStringDecryption.java> Stack string ended, since no suitable scalar value could be found at 0x10002460!
TalismanStackStringDecryption.java> --------------------------------------------------------------------------------
TalismanStackStringDecryption.java> US-ASCII: Global\DelSelf(%8.8X)
TalismanStackStringDecryption.java> ISO-LATIN-1: Global\DelSelf(%8.8X)
TalismanStackStringDecryption.java> UTF-16BE: 䜀氀漀戀愀氀尀䐀攀氀匀攀氀昀⠀─㠀⸀㠀堀⤀
TalismanStackStringDecryption.java> UTF-16LE: Global\DelSelf(%8.8X)
TalismanStackStringDecryption.java> UTF-8: Global\DelSelf(%8.8X)
TalismanStackStringDecryption.java> Length in bytes: 46
TalismanStackStringDecryption.java> --------------------------------------------------------------------------------
TalismanStackStringDecryption.java> Finished!
```

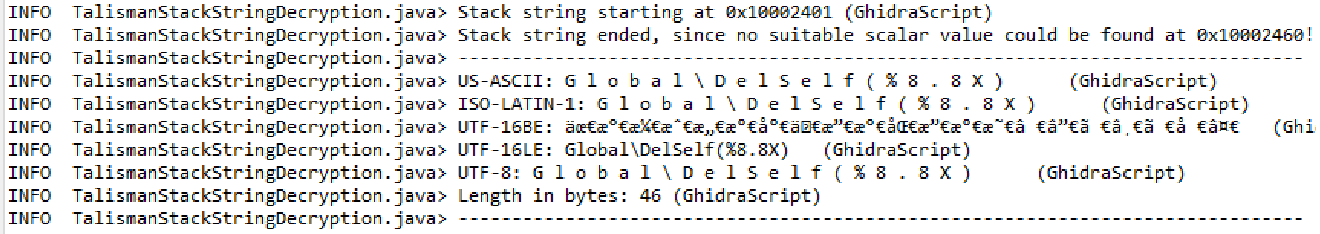
The script’s decrypted output in Ghidra’s console

When viewing the same stack strings in Eclipse’s console, the missing spaces are clearly visible, which is a clear indication that some of the stack strings are wide strings, whereas others are not.



```
INFO  TalismanStackStringDecryption.java> ------------------------------------------------------------------------------------ (Gh
INFO  StackStrings.java> Stack string starting at 0x40100d (GhidraScript)
INFO  StackStrings.java> Stack string ended, since no suitable scalar value could be found at 0x4010dd! (GhidraScript)
INFO  StackStrings.java> ---------------------------------------------------------------------------------- (GhidraScript)
INFO  StackStrings.java> US-ASCII: k e r n e l 3 2 . d l l   advapi32.dll LoadLibraryA  (GhidraScript)
INFO  StackStrings.java> ISO-LATIN-1: k e r n e l 3 2 . d l l   advapi32.dll LoadLibraryA  (GhidraScript)
INFO  StackStrings.java> UTF-16BE: æ¬€æ”€ç ˆ€æ¸€æ”€æ°€ã Œ€ã ˆ€â¸€æ

€æ°€æ°€ æ…¤ç™¡ç

©ãŒ²â¹¤æ±¬Læ½¡æ‘Œæ¥¢ç‰¡ç‰¹ä„€ (GhidraScript)
INFO  StackStrings.java> UTF-16LE: kernel32.dll æ‘¡æ…¶æ¥°ã ˆ³æ±®æ±¬ä¼€æ…¯ä±¤æ¥©æ…²ç¥²A (GhidraScript)
INFO  StackStrings.java> UTF-8: k e r n e l 3 2 . d l l   advapi32.dll LoadLibraryA  (GhidraScript)
INFO  StackStrings.java> Length in bytes: 52 (GhidraScript)
INFO  StackStrings.java> ---------------------------------------------------------------------------------- (GhidraScript)
```

The script’s output in Eclipse’s console



```
INFO  TalismanStackStringDecryption.java> Stack string starting at 0x10002401 (GhidraScript)
INFO  TalismanStackStringDecryption.java> Stack string ended, since no suitable scalar value could be found at 0x10002460!
INFO  TalismanStackStringDecryption.java> -----------------------------------------------------------------------------------
INFO  TalismanStackStringDecryption.java> US-ASCII: G l o b a l \ D e l S e l f ( % 8 . 8 X )      (GhidraScript)
INFO  TalismanStackStringDecryption.java> ISO-LATIN-1: G l o b a l \ D e l S e l f ( % 8 . 8 X )      (GhidraScript)
INFO  TalismanStackStringDecryption.java> UTF-16BE: äœ€æ°€æ¼€æˆ€æ„€æ°€å°€ä

€æ”€æ°€åŒ€æ”€æ°€æ˜€â €â”€ã €â¸€ã €å €â¤€   (Ghi
INFO  TalismanStackStringDecryption.java> UTF-16LE: Global\DelSelf(%8.8X)   (GhidraScript)
INFO  TalismanStackStringDecryption.java> UTF-8: G l o b a l \ D e l S e l f ( % 8 . 8 X )      (GhidraScript)
INFO  TalismanStackStringDecryption.java> Length in bytes: 46 (GhidraScript)
INFO  TalismanStackStringDecryption.java> -----------------------------------------------------------------------------------
```

The script’s decrypted output in Eclipse’s console

Conclusion

To conclude, the usage of encrypted stack strings is an efficient way to evade more generic detection, but does not slow analysis down much. The automation to swiftly see what the string is by selecting the string’s start in the assembly view, whose decrypted value is then printed when the script is executed via a hotkey takes very little time.

Recreating the decryption routine might consume some time, but in most cases, encrypted stack strings are generally decrypted using the same routine within a single sample. As such, the time to recreate the decryption routine is likely to end up saving time overall.

Complete script

The complete script to handle encrypted (wide) stack strings is given below. The generic stack strings script is given first, after which the PlugX Talisman script is given.

```
//A script to recreate (wide) stack strings in PlugX Talisman samples, based on
Mich's SimpleStackStrings
(https://github.com/0x6d696368/ghidra_scripts/blob/master/SimpleStackStrings.py)
//@author Max 'Libra' Kersten
//@category deobfuscation
//@keybinding
//@menupath
//@toolbar

import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

import ghidra.app.script.GhidraScript;
import ghidra.program.model.listing.Instruction;
import ghidra.program.model.scalar.Scalar;

public class StackStrings extends GhidraScript {

    @Override
    protected void run() throws Exception {
        // Get the starting instruction
        Instruction instruction = getInstructionAt(currentAddress);

        // Check if there is an instruction at the current location
        if (instruction == null) {
            // Print the error message
            println("No instruction found at 0x" +
Long.toHexString(currentAddress.getOffset()));
            // Return, thus ending the execution of the script
            return;
        }

        // If an instruction is found, print the starting offset
        println("Stack string starting at 0x" +
Long.toHexString(currentAddress.getOffset()));

        // Initialise the list to store the stack string bytes in
        List<Integer> valueBytes = new ArrayList<>();

        /*
         * Loop over the current instructions (and the ones thereafter) until no
         * instruction is found, or the loop is broken
         */
        while (instruction != null) {
            /*
             * Get the second scalar value from the instruction. When encountering
             * instructions such as "MOV dword ptr [ESP + local_var], 0x41414141",
the
             * second scalar value is the second value.
             */
            Scalar scalar = instruction.getScalar(1);

            // If the second scalar value is null, revert to the value of the first
scalar
```

```
            if (scalar == null) {
                /*
                 * Store the first scalar value, which is present instructions such
as
                 * "PUSH 0x41414141"
                 */
                scalar = instruction.getScalar(0);
                /*
                 * If there is no scalar value, the encountered instruction is not
part of the
                 * (wide) stack string
                 */
                if (scalar == null) {
                    /*
                     * Print the error, along with the address of the instruction
which contains no
                     * suitable scalar value
                     */
                    println("Stack string ended, since no suitable scalar value could
be found at 0x"
                            + Long.toHexString(instruction.getAddress().getOffset())
+ "!");
                    // Break the loop, continuing the decryption and string re-
creation process
                    break;
                }
            }

            // Get the scalar's value
            long value = scalar.getValue();
            // Get the value in hexadecimal format
            String valueString = Long.toHexString(value);

            // Add all the bytes to the list once they are converted
            valueBytes.addAll(convert(valueString));

            // Gets the next instruction
            instruction = instruction.getNext();
        }

        /*
         * Decrypt the collected bytes. If no decryption is required when re-using
this
         * script, simply convert the list of bytes into an unboxed byte array, after
         * which the existing logic will handle the rest
         */
        byte[] output = decrypt(valueBytes);

        /*
         * Create a variety of strings, based on the given bytes. If multiple stack
         * strings are present in sequence, it is possible that the strings represent
         * more than a single stack string. This is left to the analyst's
         * interpretation.
         *
         * To make the script more generic, the strings are recreated using a variety
```

```
of
         * character sets. Some stack strings might consist of wide strings, whereas
         * others might be of a more unique format. Generally, the common formats
such
         * as UTF-8 and UTF-16 will be used, as the Windows API uses those.
         */
        String usAscii = new String(output, StandardCharsets.US_ASCII);

        String isoLatin1 = new String(output, StandardCharsets.ISO_8859_1);

        String utf16be = new String(output, StandardCharsets.UTF_16BE);

        String utf16le = new String(output, StandardCharsets.UTF_16LE);

        String utf8 = new String(output, StandardCharsets.UTF_8);

        // Print all of the stack strings
        println("--------------------------------------------------------------------
-----------");
        println("US-ASCII: " + usAscii);
        println("ISO-LATIN-1: " + isoLatin1);
        println("UTF-16BE: " + utf16be);
        println("UTF-16LE: " + utf16le);
        println("UTF-8: " + utf8);
        println("Length in bytes: " + output.length);
        println("--------------------------------------------------------------------
-----------");
    }

    /**
     * Converts the given input string to a list of bytes. The input string should
     * be equal to hexadecimal values, without the leading "0x".
     *
     * @param input a string which contains the hexadecimal values, without the
     *              leading "0x"
     * @return a list of bytes, read as little endian (right from left, per byte)
     */
    private List<Integer> convert(String input) {
        // Initialise the list of bytes
        List<Integer> scalarValue = new ArrayList<>();
        // If the input is null, or empty, null is returned
        if (input == null || input.isEmpty()) {
            return null;
        }

        // Check if the length input is a multitude of two
        int remainder = input.length() % 2;

        /*
         * If the length of the string is not equal to one, add a leading zero to the
         * last character
         */
        if (remainder != 0) {
            int offset = input.length() - 1;
            String partA = input.substring(0, offset);
            String partB = input.substring(offset, offset + 1);
            input = partA + "0" + partB;
        }

        /*
         * Iterate over the string in groups of two characters at a time, which
         * represents a single byte every time. The string is iterated over starting
in
         * the end, due to the endianness
         */
        for (int i = input.length(); i > 0; i = i - 2) {
            int j = i - 2;
            // If i is zero or j is less than zero, the loop has to be broken
            if (i == 0 || j < 0) {
                break;
            }
            // Obtain the byte in string form
            String b = input.substring(j, i);
            // Convert the byte in string form to a boxed byte object
            Integer conversion = Integer.parseInt(b, 16);
            // Add the byte to the list of bytes, in sequential order
            scalarValue.add(conversion);
        }
        // Return the obtained bytes
        return scalarValue;
    }

    /*
     * A function to hold the decryption routine for the given sample, if any
     */
    private byte[] decrypt(List<Integer> input) {
        // Initialise the output variable
        byte[] output = new byte[input.size()];
        // Loop over the input and convert the given byte
        for (int i = 0; i < input.size(); i++) {
            output[i] = input.get(i).byteValue();
        }
        // Return the converted output
        return output;
    }
}
```

The PlugX Talisman stack string decryption script is given below in full.

```
//A script to recreate (wide) stack strings in PlugX' Talisman variant, based on
Mich's SimpleStackStrings
(https://github.com/0x6d696368/ghidra_scripts/blob/master/SimpleStackStrings.py)
//@author Max 'Libra' Kersten (@Libranalysis, https://maxkersten.nl)
//@category deobfuscation
//@keybinding
//@menupath
//@toolbar

import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

import ghidra.app.script.GhidraScript;
import ghidra.program.model.listing.Instruction;
import ghidra.program.model.scalar.Scalar;

public class TalismanStackStringDecryption extends GhidraScript {

    @Override
    protected void run() throws Exception {
        // Get the starting instruction
        Instruction instruction = getInstructionAt(currentAddress);

        // Check if there is an instruction at the current location
        if (instruction == null) {
            // Print the error message
            println("No instruction found at 0x" +
Long.toHexString(currentAddress.getOffset()));
            // Return, thus ending the execution of the script
            return;
        }

        // If an instruction is found, print the starting offset
        println("Stack string starting at 0x" +
Long.toHexString(currentAddress.getOffset()));

        // Initialise the list to store the stack string bytes in
        List<Integer> valueBytes = new ArrayList<>();

        /*
         * Loop over the current instructions (and the ones thereafter) until no
         * instruction is found, or the loop is broken
         */
        while (instruction != null) {
            /*
             * Get the second scalar value from the instruction. When encountering
             * instructions such as "MOV dword ptr [ESP + local_var], 0x41414141",
the
             * second scalar value is the second value.
             */
            Scalar scalar = instruction.getScalar(1);

            // If the second scalar value is null, revert to the value of the first
scalar
```

```
            if (scalar == null) {
                /*
                 * Store the first scalar value, which is present instructions such
as
                 * "PUSH 0x41414141"
                 */
                scalar = instruction.getScalar(0);
                /*
                 * If there is no scalar value, the encountered instruction is not
part of the
                 * (wide) stack string
                 */
                if (scalar == null) {
                    /*
                     * Print the error, along with the address of the instruction
which contains no
                     * suitable scalar value
                     */
                    println("Stack string ended, since no suitable scalar value could
be found at 0x"
                            + Long.toHexString(instruction.getAddress().getOffset())
+ "!");
                    // Break the loop, continuing the decryption and string re-
creation process
                    break;
                }
            }

            // Get the scalar's value
            long value = scalar.getValue();
            // Get the value in hexadecimal format
            String valueString = Long.toHexString(value);

            // Add all the bytes to the list once they are converted
            valueBytes.addAll(convert(valueString));

            // Gets the next instruction
            instruction = instruction.getNext();
        }

        /*
         * Decrypt the collected bytes. If no decryption is required when re-using
this
         * script, simply convert the list of bytes into an unboxed byte array, after
         * which the existing logic will handle the rest
         */
        byte[] output = decrypt(valueBytes);

        /*
         * Create a variety of strings, based on the given bytes. If multiple stack
         * strings are present in sequence, it is possible that the strings represent
         * more than a single stack string. This is left to the analyst's
         * interpretation.
         *
         * To make the script more generic, the strings are recreated using a variety
```

```
of
         * character sets. Some stack strings might consist of wide strings, whereas
         * others might be of a more unique format. Generally, the common formats
such
         * as UTF-8 and UTF-16 will be used, as the Windows API uses those.
         */
        String usAscii = new String(output, StandardCharsets.US_ASCII);

        String isoLatin1 = new String(output, StandardCharsets.ISO_8859_1);

        String utf16be = new String(output, StandardCharsets.UTF_16BE);

        String utf16le = new String(output, StandardCharsets.UTF_16LE);

        String utf8 = new String(output, StandardCharsets.UTF_8);

        // Print all of the stack strings
        println("--------------------------------------------------------------------
-----------");
        println("US-ASCII: " + usAscii);
        println("ISO-LATIN-1: " + isoLatin1);
        println("UTF-16BE: " + utf16be);
        println("UTF-16LE: " + utf16le);
        println("UTF-8: " + utf8);
        println("Length in bytes: " + output.length);
        println("--------------------------------------------------------------------
-----------");
    }

    /**
     * Converts the given input string to a list of bytes. The input string should
     * be equal to hexadecimal values, without the leading "0x".
     *
     * @param input a string which contains the hexadecimal values, without the
     *              leading "0x"
     * @return a list of bytes, read as little endian (right from left, per byte)
     */
    private List<Integer> convert(String input) {
        // Initialise the list of bytes
        List<Integer> scalarValue = new ArrayList<>();
        // If the input is null, or empty, null is returned
        if (input == null || input.isEmpty()) {
            return null;
        }

        // Check if the length input is a multitude of two
        int remainder = input.length() % 2;

        /*
         * If the length of the string is not equal to one, add a leading zero to the
         * last character
         */
        if (remainder != 0) {
            input = "0" + input;
        }
        /*
         * Iterate over the string in groups of two characters at a time, which
         * represents a single byte every time. The string is iterated over starting
in
         * the end, due to the endianness
         */
        for (int i = input.length(); i > 0; i = i - 2) {
            int j = i - 2;
            // If i is zero or j is less than zero, the loop has to be broken
            if (i == 0 || j < 0) {
                break;
            }
            // Obtain the byte in string form
            String b = input.substring(j, i);
            // Convert the byte in string form to a boxed integer object
            Integer conversion = Integer.parseInt(b, 16);
            // Add the byte to the list of bytes, in sequential order
            scalarValue.add(conversion);
        }
        // Return the obtained bytes
        return scalarValue;
    }

    /*
     * Decrypts the string based on the algorithm within the sample (SHA-256
     * 344fc6c3211e169593ab1345a5cfa9bcb46a4604fe61ab212c9316c0d72b0865, offset
     * 0x10002460)
     */
    private byte[] decrypt(List<Integer> input) {
        // Initialise the output variable
        byte[] output = new byte[input.size()];
        // Loop over the input and decrypt the given byte
        for (int i = 0; i < input.size(); i++) {
            output[i] = (byte) (((input.get(i) + 0x22) ^ 0x33) - 0x44);
        }
        // Return the decrypted output
        return output;
    }
}
```