Ghidra script to decrypt strings in Amadey 1.09 – Max Kersten

Archived: 2026-04-05 15:33:06 UTC

This article was published on the 9th of February 2021. This article was updated on the 8th of December 2021.

On the 21st of March 2019, the American National Security Agency (NSA) released Ghidra: a free and open-source reverse engineering tool. The tool can disassemble and decompile code for a variety of architectures. Additionally, users can create scripts in Python and Java using the exposed API. This article will cover the the string encryption in Amadey 1.09, and will provide a step-by-step guide to create an automatic string decryption script in Java.

Table of contents

The sample

This sample is taken from KrabsOnSecurity's blog from Feburary 2019. The specific sample is the unpacked stage, as is described in the blog. The sample can be downloaded from VirusBay, Malware Bazaar, or MalShare. The hashes are given below.

```
MD5: dbaaa2699c639f652117e9176fd27fdf
SHA-1: 3e4cd703deef2cfd1726095987766e2f062e9c57
SHA-256: 654b53b4ef5b98b574f7478ad11192275178ca651d9e8496070651cd6f72656a
Size: 51396 bytes
```

Additionally, snippets from this blog by Lars A. Wallenborn and Jesko H. Hüttenhain about a Ghidra script to automatically decrypt strings in REvil samples are used. Within the code, credit to Lars' and Jesko's work and websites is given whenever it is used.

Scripting basics

Scripting in Ghidra can be done using Python and Java. Python scripts are executed with the help of Jython. As such, the *native* way of scripting in Ghidra, is with Java. Therefore, this is also what will be used in this article.

In the official repository's DevGuide, guidance is given with regards to configuring Ghidra with Eclipse. This is useful to develop Java based scripts in a proper IDE, with the option to debug the script.

Once the environment is set-up, one will see that every script extends the *GhidraScript* class. This class forces the script to implement the *run* method, which is the starting point of the script's code. Due to the inheritance of the GhidraScript class, the user has access to the FlatProgramAPI. Functions might be added to this class, but never removed. Or, as the NSA explains in the accompanied JavaDoc:

```
This class is a flattened version of the Program API.

NOTE:

    1. NO METHODS SHOULD EVER BE REMOVED FROM THIS CLASS.
    2. NO METHOD SIGNATURES SHOULD EVER BE CHANGED IN THIS CLASS.

This class is used by GhidraScript.

Changing this class will break user scripts.

That is bad. Don't do that.
```

Using Eclipse's built-in auto-completion and JavaDoc viewer, its easy to view all functions that are directly accessible from the GhidraScript class. Alternatively, or additionally, one can read the publicly available JavaDoc on the NSA's website.

As is specified in the GhidraScript class, there are several variables that one can access within a script, without the need to initialise them. An excerpt of the Ghidra documentation is given below.

```
All scripts, when run, will be handed the current state in the form of class instance variable. Thes

currentProgram: the active program
currentAddress: the address of the current cursor location in the tool
currentLocation: the program location of the current cursor location in the tool, or null if no prog
currentSelection: the current selection in the tool, or null if no selection exists
currentHighlight: the current highlight in the tool, or null if no highlight exists
```

Knowing the basics of scripting before starting to write code will result in a more efficient use of your time, as well as cleaner code.

Outline

The goal of the script is to automatically decrypt the encrypted strings that are present within the binary. As such, it is essential to be able to decrypt the strings. For this reason, it is the first step in this article. After that, it is important to get the required information from the user. What that is precisely, will follow logically from the decryption routine.

Knowing how to decrypt the strings is only part of the job, as one will also need to find all references to the decryption function call and its argument(s). Knowing the arguments for each call, and the decryption routine itself, will allow the script to decrypt all strings. Knowing how to add comments to the disassembly and decompiler view will show the result in an organised fashion to the analyst.

Before putting all the pieces are put together, a method to cache decrypted strings will be introduced. This will reduce the time the script needs to run if the same string is encountered multiple times.

In this article, a Ghidra build from the 21st of December 2020 (which has a few more commits than Ghidra 9.2.1) is used. Due to the usage of the FlatProgramAPI, it should run on future versions, and is likely to run on older versions.

Finding the decryption routine

After loading this sample in Ghidra and running the default analysers, it becomes apparent that the sample's symbols were not stripped during the compilation. As such, it becomes easy to find the main function, as is given below. Note that the function is called *_main*.

```
int __cdecl _main(int _Argc,char **_Argv,char **_Env)
{
  char *pcVar1;
  size_t in_stack_fffffff0;

  __alloca(in_stack_fffffff0);
  ___main();
  __Z10aBypassUACv();
  pcVar1 = __Z12aGetSelfPathv();
  __Z13aDropToSystemPc(pcVar1);
  pcVar1 = __Z19aGetSelfDestinationi(0);
  __Z11aAutoRunSetPc(pcVar1);
  __Z6aBasici(0);
  return 0;
}
```

The mangled names provide insight into what the functions do. These symbols can be misleading, and should not always be trusted as-is, but the symbols in this sample are representative for the functionality that is within the functions. The *___main* function, note the triple underscore, does not contain code that was made by the author.

Searching for encrypted strings can be done in a variety of ways. One can use the *Defined Strings* view (as found in the *Window* toolstrip menu), browse through the function tree to look for a name that is likely to handle

encrypted strings (which is *__Z8aDecryptPc*), or click through the functions until one encounters a function call that seems to decrypt a string (as can be seen in *__Z6aBasici*).

The decryption routine is given below.

```
undefined * __cdecl __Z8aDecryptPc(char *param_1)
{
  size_t sVar1;
  uint local_10;

  _memset(&_ZZ8aDecryptPcE14aDecryptResult,0,0x400);
  local_10 = 0;
  while( true ) {
    sVar1 = _strlen(param_1);
    if (sVar1 <= local_10) break;
    sVar1 = _strlen(s_1ee76e11929a07445c5abd744aa407db_00405000);
    (&_ZZ8aDecryptPcE14aDecryptResult)[local_10] =
         param_1[local_10] - s_1ee76e11929a07445c5abd744aa407db_00405000[local_10 % sVar1];
    local_10 = local_10 + 1;
  }
  return &_ZZ8aDecryptPcE14aDecryptResult;
}
```

At first, two variables are declared, after which the memory buffer for the output is set. The variable *local_10* is incremented with one at the end of every iteration in the while loop. Only when the length of the input is equal or bigger than the amount of iterations that have taken place, the endless loop breaks. At last, the result is returned. The decryption itself is based on the used key, together with the iterative value. Refactoring the method with more readable names, the function becomes easily readable.

```
undefined * __cdecl __Z8aDecryptPc(char *input)
{
  size_t inputLength;
  uint i;

  _memset(&result,0,0x400);
  i = 0;
  while( true ) {
    inputLength = _strlen(input);
    if (inputLength <= i) break;
    inputLength = _strlen(key);
    (&result)[i] = input[i] - key[i % inputLength];
    i = i + 1;
  }
  return &result;
}
```

In conclusion, this function requires one argument, which is decrypted using a hardcoded key, after which the decrypted value is returned.

Remaking the decryption routine

As the Ghidra script is to be written in Java, the decryption routine is also to be written in Java. The ported function will take two arguments, rather than one. The first one is the input, which is the string to decrypt. As memory is read from the sample, the type of this variable is a byte array. The second argument is the decryption key, which is represented as a string. The ported function is more readable than the decompiled code, as can be seen below.

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

        for (int i = 0; i < input.length; i++) {
                output[i] = (byte) (input[i] - keyArray[i % keyLength]);
        }

        return new String(output);
}
```

Getting user-input

Getting information from the user is a useful way to make a script more generic. If a specific malware family uses the same decryption routine with a different key per sample, one can request the key from the user, without the need to alter the decryption script.

In Ghidra, one can request values from the user using the *ask** functions, where the asterisk should be read as a wildcard, as there are many functions present to help. Aside from the added convenience of not having to write such a function, it is important to note that a user cannot provide an empty string to this dialog. The requested string cannot be *null* either, as closing the dialog will lead to the termination of the script, which is clearly shown to the user in Ghidra's console.

To print data to the console, one can use the built-in *println* and *print* functions. The difference is that the latter does not print the script name into the console.

In this case, the *askString* function is used to request a string from the user. Within the script, two values will be required: the name of the decryption function and the key that is used to decrypt the encrypted input.

Getting all cross references for the decryption function

The decryption function's name is obtained earlier on in the script, as the user provides the name. Based on that, one can get a list of functions that use this name. As symbol names do not have to be unique in Ghidra, it is

possible that there are more functions with the same name. The code to obtain such a list is given below.

```
List<Function> functions = getGlobalFunctions(functionName);
```

Basic sanity checks to see if there are more functions with the given name can be implemented. The *ReferenceIterator* class is present in the *currentProgram* variable, which is already initialised. Using the *getReferencesTo* function, one can use an *Address* object to get all references to that address. To convert a raw address, which is represented as a *Long*, to an *Address* object, one can use the *toAddr* function. This function is accessible via the extended *GhidraScript* class.

```
ReferenceIterator references = currentProgram.getReferenceManager().getReferencesTo(toAddr(decryptio
```

To iterate over all references, one can use a simple for-loop, as is shown below.

```
for (Reference reference : references) {
    Address address = reference.getFromAddress();
    //...
}
```

This for-loop is the basis for the following steps, as these have to be done per reference. Obtaining the address for the reference is the first action that has to be completed.

Iterating all decryption calls

This part of the script is based upon two steps. The first one is being able to decrypt a given string, which is possible due to the decryption method that was written in an earlier step. The second step is to obtain the encrypted string for each function call. To do so, one can use code from the earlier mentioned blog by Lars A. Wallenborn and Jesko H. Hüttenhain.

The function named *getConstantCallArgument* is used. This function requires two arguments, the first being an *Address* object, and the second is an integer array. The *Address* object is the address of the function call. The integer array is used to obtain one or more arguments of the given function's call. The indices of the arguments in this array correspond with the arguments for the function call, where the first index is *1*, unlike the usual *0* in an array. The function is given below.

```
//Code by Lars A. Wallenborn and Jesko H. Hüttenhain (see https://blag.nullteilerfrei.de/2020/02/02/
private OptionalLong[] getConstantCallArgument(Address addr, int[] argumentIndices)
                throws IllegalStateException, IllegalArgumentException {
        int argumentPos = 0;
        OptionalLong argumentValues[] = new OptionalLong[argumentIndices.length];
        Function caller = getFunctionBefore(addr);
        if (caller == null)
                throw new IllegalStateException();
```

```
        DecompInterface decompInterface = new DecompInterface();
        decompInterface.openProgram(currentProgram);
        DecompileResults decompileResults = decompInterface.decompileFunction(caller, 120, monitor);
        if (!decompileResults.decompileCompleted())
                throw new IllegalStateException();
        HighFunction highFunction = decompileResults.getHighFunction();
        Iterator<PcodeOpAST> pCodes = highFunction.getPcodeOps(addr);
        while (pCodes.hasNext()) {
                PcodeOpAST instruction = pCodes.next();
                if (instruction.getOpcode() == PcodeOp.CALL) {
                        for (int index : argumentIndices) {
                                argumentValues[argumentPos] = traceVarnodeValue(instruction.getInput
                                argumentPos++;
                        }
                }
        }
        return argumentValues;
}
```

This function returns an array of *OptionalLong* objects. Such an object can contain a *Long*, although it might not. In some cases, there might occur an error whilst retrieving the address, meaning that the object itself is actually set to *null*. To avoid returning *null*, the *OptionalLong* object is used.

Within the function, the decompiler interface is used to get access to the PCode values. For each call, the *Varnode*'s value is traced using *traceVarnodeValue*, which is also written by Lars A. Wallenborn and Jesko H. Hüttenhain. The code is given below.

```
//Code by Lars A. Wallenborn and Jesko H. Hüttenhain (see https://blag.nullteilerfrei.de/2020/02/02/
private OptionalLong traceVarnodeValue(Varnode argument) throws IllegalArgumentException {
        while (!argument.isConstant()) {
                PcodeOp ins = argument.getDef();
                if (ins == null)
                        break;
                switch (ins.getOpcode()) {
                case PcodeOp.CAST:
                case PcodeOp.COPY:
                        argument = ins.getInput(0);
                        break;
                case PcodeOp.PTRSUB:
                case PcodeOp.PTRADD:
                        argument = ins.getInput(1);
                        break;
                case PcodeOp.INT_MULT:
                case PcodeOp.MULTIEQUAL:
                        return OptionalLong.empty();
                default:
```

```
                        throw new IllegalArgumentException(String.format("Unknown opcode %s for vari
                                        ins.getMnemonic(), argument.getAddress().getOffset()));
                }
        }
        return OptionalLong.of(argument.getOffset());
}
```

Within the decryption routine in Amadey 1.09, only a single argument is used. This argument points to an encrypted value of a string. As such, the index one wants to retrieve is the only equal to one, meaning an integer array with the value *1* at index *0* is required as input. The address for the function is the address of each referenced function call of the decryption routine. The integer array is given below.

```
int[] argumentIndices = { 1 };
```

One can get the *Long* value from the *OptionalLong* object by using the *getAsLong* function, as can be seen in the code below.

```
Long argument = arguments[0].getAsLong();
```

Right now, all information to decrypt a string has been obtained, as the address of the encrypted string, the decryption key, and the decryption routine have been collected. The *getDecryptedArgument* function contains all code to decrypt an argument, which is then returned as a string. It requires an address as a *Long*, and the decryption string as a *String*. The code for the function is given below.

```
private String getDecryptedArgument(Long argument, String key) throws MemoryAccessException {
        MemoryBlock block = getMemoryBlock(toAddr(argument));
        int size = ((Long) block.getSize()).intValue();
        byte[] input = getBytes(toAddr(argument), size);
        String decryptedValue = decrypt(input, key);
        decryptedValue = decryptedValue.replace("\n", "\\n").replace("\r", "\\r");
        return getFirstReadableString(decryptedValue);
}
```

At first, a memory block is read, based on the address of the given argument. The *getMemoryBlock* function is accessible via the *GhidraScript* class. The size of the block is stored in a different variable to increase the readability of the code. The *getBytes* function, also accessible via the *GhidraScript* class, gets the bytes from the given argument's location until the location plus the given size.

The *decryptedValue* string contains the decrypted string. If any newline and carriage return values are present in that string, they are escaped using the chained *replace* function calls. The input that was provided to the decryption function is much bigger than the actual string. As such, the first human readable string has to be recovered, which is done using the *getFirstReadableString* method.

This method, as can be seen below, requires a string as input, and will return the first human readable string.

```
private String getFirstReadableString(String input) {
        int beginIndex = -1;
        int endIndex = -1;

        int asciiLow = 0;
        int asciiHigh = 255;

        for (int i = 0; i < input.toCharArray().length; i++) {
                char currentChar = input.charAt(i);
                if (currentChar < asciiHigh || currentChar > asciiLow) {
                        beginIndex = i;
                        break;
                }
        }

        for (int i = 0; i < input.toCharArray().length; i++) {
                char currentChar = input.charAt(i);
                if (currentChar > asciiHigh || currentChar < asciiLow) {
                        endIndex = i;
                        break;
                }
        }

        if (beginIndex >= 0 && endIndex >= 0) {
                return input.substring(beginIndex, endIndex);
        }

        return "NO_ASCII_STRING_FOUND";
}
```

This function declares four local integers, named *beginIndex*, *endIndex*, *asciiLow*, and *asciiHigh*. The first two are set to *-1*, whereas the last two are set to *0* and *255* respectively. The *beginIndex* and *endIndex* are used to store the beginning and ending indices of the first human readable string, whereas the latter two variables are hardcoded to define the beginning and ending of the human readable range of ASCII characters. Wide strings are out of scope for this function, as they are not used within this sample.

The given string is iterated over twice: once for the first character, and once for the last character. Whilst this can be done in a single loop, this would decrease the code's readability. As this function is rather rudimentary at best, the code's optimisation has not been included in this article.

The return value of this function is a substring of the provided input. If none of the characters are readable, then a default value is returned. This default value is never returned in the used sample.

Caching the decrypted results

Even though some optimisation steps were left out in the previous step, this step is purely meant as an optimisation. Caching results is a useful way to easily decrease the time that the script takes, without adding a needless complex layer of logic into the code.

At first, a mapping is created. Mappings are often known as dictionaries in other languages. In this case, the mapping will use addresses (as a *Long*) as a key, where the value at a given key is a *String*. The mapping's keys are the locations of the encrypted variables, whereas the mapping's values are decrypted strings. The creation of the mapping is given below.

```
Map<Long, String> handled = new HashMap<>();
```

When iterating over the argument locations, the following if-statement is required to implement the caching of decrypted values.

```
if (handled.containsKey(argument)) {
    //...
} else {
    //...
}
```

As such, it checks if the given address already present in the mapping. If it is, the value for the given key should be used. Otherwise, it can go through the normal decryption process, and add the outcome to the given mapping. The lookup in the mapping is much quicker than the decryption routine, thereby saving several seconds in a small sample such as this. In testing on my local machine, the script's runtime went down from 22 seconds to 16 seconds. Slower computers might benefit more of the caching, whereas faster computers might benefit less.

Adding comments and bookmarks

Once the decrypted values have been obtained, they need to be handed back to the user. This is done in multiple ways. One can add comments (both to the disassembly and decompiler views), bookmarks, and print the results in the output window. In the code below, both comment methods, as well as the bookmark creation, have been listed. Do note that existing bookmarks for a specific address will be overwritten with the *createBookmark* function.

```
//Decompiler comment
currentProgram.getListing().getCodeUnitAt(toAddr(argument)).setComment(CodeUnit.PLATE_COMMENT, comme
//Disassembly comment
currentProgram.getListing().getCodeUnitAt(toAddr(argument)).setComment(CodeUnit.PRE_COMMENT, comment
//Bookmark creation
createBookmark(toAddr(argument), "Decrypted string", "The variable named " + getSymbolAt(toAddr(argu
```

Putting it all together

The complete script is given below. It contains a few more sanity checks, the most notable being the user feedback when providing details, and the check if there is only a single function with the provided name. When running the script, the following output is observed at the end, aside from the list of decrypted values in Ghidra's console:

```
Decrypted 47 strings (using 11 cached strings), placed 210 comments, and
created 47 bookmarks in 16 seconds!
```

To show the difference in the decompiled code, two excerpts are given. The first excerpt is the decompiler output in Ghidra prior to running the string decryption script, as can be seen below.

```
pcVar2 = __Z8aDecryptPc(&_aAV00);
bVar1 = __Z7aPathAVPc(pcVar2);
uVar3 = bVar1 != false;

pcVar2 = __Z8aDecryptPc(&_aAV01);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 2;
}

pcVar2 = __Z8aDecryptPc(&_aAV02);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 3;
}

pcVar2 = __Z8aDecryptPc(&_aAV03);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 4;
}

pcVar2 = __Z8aDecryptPc(&_aAV04);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 5;
}
```

The second one contains the comments that have been added by the script.

```
                  /* Decrypted value: "AVAST Software" */
pcVar2 = __Z8aDecryptPc(&_aAV00);
bVar1 = __Z7aPathAVPc(pcVar2);
uVar3 = bVar1 != false;
                  /* Decrypted value: "Avira" */
```

```
pcVar2 = __Z8aDecryptPc(&_aAV01);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 2;
}
                    /* Decrypted value: "Kaspersky Lab" */
pcVar2 = __Z8aDecryptPc(&_aAV02);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 3;
}
                    /* Decrypted value: "ESET" */
pcVar2 = __Z8aDecryptPc(&_aAV03);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 4;
}
                    /* Decrypted value: "Panda Security" */
pcVar2 = __Z8aDecryptPc(&_aAV04);
bVar1 = __Z7aPathAVPc(pcVar2);
if (bVar1 != false) {
  uVar3 = 5;
}
```

The decompiled code becomes easily readable, especially with the included symbols. This allows the analyst to quickly analyse the malware's functionality.

Conclusion

Scripting will greatly reduce the amount of time that an analyst needs to spend when looking at a repetitive task. Understanding the basics of Ghidra's architecture will greatly reduce the amount of time an analyst needs to write such a script. By creating a heavily documented script, it becomes reusable for similar tasks in other samples. As such, each new script will make future work easier.

Note that not all parts of the script in this article are required, more specifically the caching part. Even though it does save some time in this script, the overhead that it requires to write the code is longer than the time that is saved with it. When looking at a sample that has a really heavy decryption routine, and/or a lot of encrypted strings, the overhead might be worth it.

This trade-off is up to the analyst to decide, and is also depending on the goal of analyst. Learning how to write such a script might involve more work that is not necessarily the most efficient for a given sample, but has a positive influence on the analyst's future scripting abilities.

To contact me, you can e-mail me at [info][at][maxkersten][dot][nl], or DM me on BlueSky @maxkersten.nl.

The complete script

The complete script is given below. It can be added as a file to any working *ghidra_script* directory, or one can use the simple editor within Ghidra to create a new script and paste this script in it.

```
//This script is used to annotate function calls to decrypt strings with the decrypted string. The d
//@author Max 'Libra' Kersten
//@category String decryption
//@keybinding
//@menupath
//@toolbar

import java.time.Duration;
import java.time.Instant;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.OptionalLong;

import ghidra.app.decompiler.DecompInterface;
import ghidra.app.decompiler.DecompileResults;
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.listing.CodeUnit;
import ghidra.program.model.listing.Function;
import ghidra.program.model.mem.MemoryAccessException;
import ghidra.program.model.mem.MemoryBlock;
import ghidra.program.model.pcode.HighFunction;
import ghidra.program.model.pcode.PcodeOp;
import ghidra.program.model.pcode.PcodeOpAST;
import ghidra.program.model.pcode.Varnode;
import ghidra.program.model.symbol.Reference;
import ghidra.program.model.symbol.ReferenceIterator;

public class Amadey extends GhidraScript {

        /**
         * Global variable that is used to keep track of the amount of decrypted strings
         */
        private int decryptionCount;

        /**
         * Global variable that is used to keep track of the amount of comments that
         * have been set
         */
```

```
	private int commentCount;

	/**
	 * Global variable that is used to keep track of the amount of bookmarks that
	 * have been set
	 */
	private int bookmarkCount;

	/**
	 * Global variable that is used to keep track of the amount of variables that
	 * have been cached
	 */
	private int cacheCount;

	/**
	 * This function is the first that is called by Ghidra when running the script.
	 * In here, variables need to be initialised first. This script uses this
	 * function to obtain information from the user, and to verify this information.
	 * After the verification and other sanity checks, the "handle" function is
	 * called. In there, the decryption logic starts.
	 */
	public void run() throws Exception {
		// Initialisation of the three counters, starting at zero
		decryptionCount = 0;
		commentCount = 0;
		bookmarkCount = 0;
		cacheCount = 0;

		/*
		 * Requesting the function name in a pop-up dialog. The result is provided as
		 * the return value, which is stored into the variable
		 */
		String functionName = askString("Function name required",
				"Please provide the name of the function that decrypts the strings:"
		// Provide feedback to the user regarding the provided function name
		println("Received function name: " + functionName);

		/*
		 * Request the decryption key in a pop-up dialog. The result is provided as the
		 * return value, which is stored into the variable
		 */
		String key = askString("Decryption key required",
				"Please provide the name of the function that is used during the str
		// Provide feedback to the user regarding the provided decryption key
		println("Received decryption key: " + key);

		/*
```

```
 * The lack of checks for null values or empty strings are omitted, as the
 * pop-up dialog does not accept empty strings. Closing the dialog will result
 * in the cancellation of the whole script, which is out of scope to handle
 * here.
 */

/*
 * Get all the functions for the given name. Symbol names do not need to be
 * unique in Ghidra, hence the fact that a list is returned
 */
List<Function> functions = getGlobalFunctions(functionName);

// If there is only one function with that name, it is safe to continue
if (functions.size() == 1) {
        // The first, and only, function resides at index 0
        Function function = functions.get(0);
        /*
         * The function address is the first address of the function, which is the
         * minimum.
         */
        long decryptionFunctionAddress = function.getBody().getMinAddress().getOffse
        // Provide feedback to the user about the decryption function details
        println("Decryption function (" + function.getName() + " found at 0x"
                        + Long.toHexString(decryptionFunctionAddress) + ")");
        // A divider is printed as the decryption process is about to start
        println("-----------------------------------------------------------------")

        // To keep track of the time, the start time is saved
        Instant begin = Instant.now();
        // The "handle" function deals with the decryption logic
        handle(decryptionFunctionAddress, key);
        // After the function returns, the end time is saved
        Instant end = Instant.now();
        // The difference between the two is calculated
        Duration duration = Duration.between(begin, end);
        // As the decryption routine has finished, a new divided is printed
        println("-----------------------------------------------------------------")

        // The collected statistics, as well as the time the script took, are then
        // printed
        println("Decrypted " + decryptionCount + " strings (using " + cacheCount + "
                        + commentCount + " comments, and created " + bookmarkCount +
                        + " seconds!");
        // Provide more information about the script to the user
        println("If you have any questions or suggestions, feel free to ping me on Tw
} else if (functions.size() == 0) {
        /*
```

```
                 * If no function is found, the user is notified and the script returns
                 */
                println("No functions were found for the given name, please make sure the nam
        } else if (functions.size() >= 2) {
                /*
                 * If multiple functions are using the same name, the user should pick a uni
                 * name for the decryption function and try again
                 */
                println("More than one function with this name has been found. Ensure that t
        }
}

/**
 * This function handles the string decryption logic. It caches the decrypted
 * strings, meaning strings that are decrypted more than once, do not need to be
 * decrypted, as the mapping already exists. The decrypted variables and
 * cross-references are commented in both the disassembly view, as well as the
 * decompiler. Additionally, bookmarks are added for each decrypted string.
 *
 * @param decryptionFunctionAddress the address of the decryption function
 *                                  within the sample
 * @param key                       the decryption key
 */
private void handle(long decryptionFunctionAddress, String key) {
        /*
         * Create a mapping for handled strings and the address of the variable the
         * encrypted content resides at
         */
        Map<Long, String> handled = new HashMap<>();
        /*
         * The reference iterator is not limited to a reference amount limit, whereas
         * some other methods are
         */
        ReferenceIterator references = currentProgram.getReferenceManager()
                        .getReferencesTo(toAddr(decryptionFunctionAddress));

        // Iterate over all references
        for (Reference reference : references) {
                /*
                 * Get the address of the location that calls the string decryption function
                 */
                Address address = reference.getFromAddress();

                /*
                 * Get the index of the argument that is to be decrypted. Note that the inde
                 * count for this starts at 1 (unlike the usual starting point of 0).
                 */
```

```
		int[] argumentIndices = { 1 }; // The decryption routine has only one argume

		try {
			// Get an array with the addresses of the given indices, in this cas
			OptionalLong[] arguments = getConstantCallArgument(address, argument
			// Get the address from the array as a long
			Long argument = arguments[0].getAsLong();

			// Initialise the variables to ensure there are no compiler errors
			String decryptedValue = "";
			String comment = "";
			/*
			 * If the mapping already contains the address of the encrypted vari
			 * already been encountered (and thus decrypted) before. Simply obta
			 * value in the mapping for the given key (the variable's address) y
			 * correct result and decreases the time the script needs to run
			 */
			if (handled.containsKey(argument)) {
				decryptedValue = handled.get(argument);
				comment = "Decrypted value (from cache): \"" + decryptedValu
				//Increase the cache count with one
				cacheCount++;
			} else {
				/**
				 * If the address of the encrypted variable is not present,
				 * encountered before. As such, it needs to be decrypted.
				 */
				decryptedValue = getDecryptedArgument(argument, key);
				comment = "Decrypted value: \"" + decryptedValue + "\"";

				/*
				 * Set comments at the variable itself in the disassembly an
				 * respectively
				 */
				currentProgram.getListing().getCodeUnitAt(toAddr(argument)).
							comment);
				currentProgram.getListing().getCodeUnitAt(toAddr(argument)).
							comment);
				// Increase the comment count with two, based on the above a
				commentCount += 2;

				// Create a bookmark at the encrypted variable's address
				createBookmark(toAddr(argument), "Decrypted string", "The va
							+ getSymbolAt(toAddr(argument)) + " is equal

				// Increase the bookmark count, based on the above added boo
				bookmarkCount++;
```

```
                                // Add this address (and the decrypted value) to the mapping
                                handled.put(argument, decryptedValue);
                        }

                        /*
                         * Regardless how the data was obtained, the user is provided with f
                         * related to decryption of the string. The argument is also printed
                         * becomes clickable in the console in Ghidra, allowing the user to
                         * it by double clicking.
                         */
                        println(comment + " (located at 0x" + Long.toHexString(argument) + "

                        /*
                         * Set comments at the reference in the disassembly and decompiler v
                         * respectively
                         */
                        currentProgram.getListing().getCodeUnitAt(reference.getFromAddress()
                                        comment);
                        currentProgram.getListing().getCodeUnitAt(reference.getFromAddress()
                                        comment);
                        // Increase the comment count with two, based on the above added com
                        commentCount += 2;
                } catch (Exception ex) {
                        println(ex.getMessage());
                }
        }

}

/**
 * Gets the decrypted argument from the given address, which is decrypted using
 * the given key
 *
 * @param argument the address of the argument
 * @param key      the key that is used to decrypted the value at the argument
 * @return the decrypted string
 * @throws MemoryAccessException if an error occurs when obtaining the bytes at
 *                               the given address
 */
private String getDecryptedArgument(Long argument, String key) throws MemoryAccessException
        // Gets the memory block at the argument's address
        MemoryBlock block = getMemoryBlock(toAddr(argument));
        // The size of the memory block
        int size = ((Long) block.getSize()).intValue();
        // Get the bytes at the given address for the given size
        byte[] input = getBytes(toAddr(argument), size);
        // Get the raw string from the decryption routine
```

```
		String decryptedValue = decrypt(input, key);
		/*
		 * Replace values that are lower than 32 in the ASCII table with escaped ones,
		 * only those which are used in this sample are visible here
		 */
		decryptedValue = decryptedValue.replace("\n", "\\n").replace("\r", "\\r");
		// Get the first readable string from the block of memory and return that value
		return getFirstReadableString(decryptedValue);
	}

	/**
	 * Gets the first readable ASCII string from a given input. If no such value can
	 * be found, the function will return "NO_ASCII_STRING_FOUND".
	 *
	 * @param input the string to obtain the first readable ASCII string from
	 * @return the first readable ASCII string
	 */
	private String getFirstReadableString(String input) {
		// The begin index of the human readable ASCII string
		int beginIndex = -1;
		// The end index of the human readable ASCII string
		int endIndex = -1;

		// The lowest human readable ASCII value
		int asciiLow = 0;
		// The highest human readable ASCII value
		int asciiHigh = 255;

		/*
		 * This loop iterates over the given string to find the first human readable
		 * ASCII character. When it does, the beginIndex variable is set to that value
		 * and the loop is broken.
		 */
		for (int i = 0; i < input.toCharArray().length; i++) {
			char currentChar = input.charAt(i);
			if (currentChar < asciiHigh || currentChar > asciiLow) {
				beginIndex = i;
				break;
			}
		}

		/*
		 * This loop iterates over the given string to find the last human readable
		 * ASCII character. When it does, the endIndex variable is set to that value and
		 * the loop is broken.
		 */
		for (int i = 0; i < input.toCharArray().length; i++) {
```

```
                char currentChar = input.charAt(i);
                if (currentChar > asciiHigh || currentChar < asciiLow) {
                        endIndex = i;
                        break;
                }
        }

        /*
         * If the beginIndex and endIndex are equal to zero or more, both values have
         * been found in the two loops. As such, the human readable substring can be
         * returned as a substring from the input at the given two indices.
         */
        if (beginIndex >= 0 && endIndex >= 0) {
                return input.substring(beginIndex, endIndex);
        }

        /*
         * If either (or both) of the indices could not be found, the default value is
         * returned
         */
        return "NO_ASCII_STRING_FOUND";
}

/**
 * The decryption routine that is present in this specific sample
 *
 * @param input the encrypted data
 * @param key   the decryption key
 * @return the decrypted string
 */
private String decrypt(byte[] input, String key) {
        // The key from the sample
        char[] keyArray = key.toCharArray();
        // The length of the key in the sample
        int keyLength = keyArray.length;
        // The encrypted data is stored in input
        // The decrypted data
        byte[] output = new byte[input.length];

        // Loop through the input
        for (int i = 0; i < input.length; i++) {
                // Decrypt the character
                output[i] = (byte) (input[i] - keyArray[i % keyLength]);
        }

        // Increase the decryption count
        decryptionCount++;
```

```
		// Return the decrypted string
		return new String(output);
	}

	/**
	 * This function returns an array of optional longs. The size of this array is
	 * equal to the size of the argument indices' size. The indices of the functoin
	 * at the given address starts at 1, unlike the usual 0.
	 *
	 * @author Lars A. Wallenborn and Jesko H. Hüttenhain(see
	 *         https://blag.nullteilerfrei.de/2020/02/02/defeating-sodinokibi-revil-string-obfus
	 *         with a slight change in the exception handling by Max 'Libra' Kersten
	 *
	 * @param addr            the address of the function
	 * @param argumentIndices the indices of the arguments of said function,
	 *                        starting at 1
	 * @return an array of optional longs, with the addresses of the variables
	 * @throws IllegalStateException if the previously defined function is null or
	 *                               if the decompiler fails to complete
	 * @throws UnknownVariableCopy   if the variable's varnode type is unknown
	 */
	private OptionalLong[] getConstantCallArgument(Address addr, int[] argumentIndices)
			throws IllegalStateException, IllegalArgumentException {
		int argumentPos = 0;
		OptionalLong argumentValues[] = new OptionalLong[argumentIndices.length];
		Function caller = getFunctionBefore(addr);
		if (caller == null)
			throw new IllegalStateException();
		DecompInterface decompInterface = new DecompInterface();
		decompInterface.openProgram(currentProgram);
		DecompileResults decompileResults = decompInterface.decompileFunction(caller, 120, m
		if (!decompileResults.decompileCompleted())
			throw new IllegalStateException();
		HighFunction highFunction = decompileResults.getHighFunction();
		Iterator<PcodeOpAST> pCodes = highFunction.getPcodeOps(addr);
		while (pCodes.hasNext()) {
			PcodeOpAST instruction = pCodes.next();
			if (instruction.getOpcode() == PcodeOp.CALL) {
				for (int index : argumentIndices) {
					argumentValues[argumentPos] = traceVarnodeValue(instruction.
					argumentPos++;
				}
			}
		}
		return argumentValues;
	}
```

```
        /**
         * This function returns an optional long for the given argument's value.
         *
         * @author Lars A. Wallenborn and Jesko H. Hüttenhain (see
         *         https://blag.nullteilerfrei.de/2020/02/02/defeating-sodinokibi-revil-string-obfus
         *         with a slight change in the exception handling by Max 'Libra' Kersten
         *
         * @param argument the instruction at the given index as a Varnode object
         * @return the address of the argument's value
         * @throws UnknownVariableCopy if the variable is unknown
         */
        private OptionalLong traceVarnodeValue(Varnode argument) throws IllegalArgumentException {
                while (!argument.isConstant()) {
                        PcodeOp ins = argument.getDef();
                        if (ins == null)
                                break;
                        switch (ins.getOpcode()) {
                        case PcodeOp.CAST:
                        case PcodeOp.COPY:
                                argument = ins.getInput(0);
                                break;
                        case PcodeOp.PTRSUB:
                        case PcodeOp.PTRADD:
                                argument = ins.getInput(1);
                                break;
                        case PcodeOp.INT_MULT:
                        case PcodeOp.MULTIEQUAL:
                                return OptionalLong.empty();
                        default:
                                throw new IllegalArgumentException(String.format("Unknown opcode %s
                                                ins.getMnemonic(), argument.getAddress().getOffset()
                        }
                }
                return OptionalLong.of(argument.getOffset());
        }

}
```

Source: https://maxkersten.nl/binary-analysis-course/analysis-scripts/ghidra-script-to-decrypt-strings-in-amadey-1-09/