

Defeating Sodinokibi/REvil String-Obfuscation in Ghidra – nullteilerfrei

By born

Archived: 2026-04-05 20:14:04 UTC

This post describes the memory layout as well as the method used by the Sodinokibi (or REvil) ransomware to protect its strings. It will then list a few Java snippets to interact with the Ghidra scripting API and finally explain a working script to deobfuscate all strings within a REvil sample. If you don't care about the explanation, you can find the [most recent version of the script you can simply import into Ghidra on github](#).

Note: All code in this blog post was written by Lars Wallenborn and Jesko Hüttenhain. It is published under the [3-Clause BSD License](#).

Some Thank-You Notes

I'd like to thank the lovely people at OALabs to bring this sample to my attention once again! If you haven't notice, definitely take a look at the [automated unpacking service UnpacMe](#). They just released it into Beta and it is working perfectly for me so far. You can find a [very expensive commercial for their unpacking service](#) in the references of this blog post. Oh and that video also contains a full analysis of the string obfuscation in REvil including the same things I'm doing in this blog post but for IDA.

A tip to the hat towards Thomas Roth, who almost certainly doesn't know me but published a lot of details about the Ghidra scripting API. And, last but not least, thanks to my buddy [Jesko](#) for figuring out Pcode traversal with me.

Memory Layout

The Sodinokibi (or REvil) ransomware leverages string obfuscation to hinder analysis ((more precisely: to slow down a bottom-up approach starting with interesting strings)). Like [all the other coolkids](#), we will use the sample with SHA256 hash

```
5f56d5748940e4039053f85978074bde16d64bd5ba97f6f0026ba8172cb29e93
```

as an example. In uses calls like the following to deobfuscated strings before usage:

```
FUN_00404e03(8DAT_0041c040, 0x256, 6, 8, local_14);
```

The call involves five arguments: The first argument `DAT_0041c040` is the same for all calls and references a global buffer containing data that doesn't look like anything. The next argument `0x256` is an offset into that buffer and the next two arguments `6` and `8` are length values. The fifth and last parameter `local_14` is a variable that will contain a deobfuscated string after the function returns.

The two length values specified lengths of a key buffer and of buffer containing encrypted data respectively. These two buffers are located consecutively directly after the specified offset in the referenced global buffer.

Decryption Algorithm

The malware uses the RC4 algorithm to decrypt the obfuscated string with the above-described key. As always, this algorithm can easily be identified by three consecutive loops where the first loop initializes the cells of an array of length 256 by their indices, the second references to key and the third references the encrypted data. The following is a slightly annotated version Ghidra's decompiled output for the RC4 algorithm:

```
do {
    sbox[k] = (byte)k;
    k = k + 1;
} while (k < 0x100);

k = 0
do {
    Tmp = sbox[k];
    i = Tmp + Key[k % KeyLength] + i & 0xff;
    sbox[k] = sbox[i];
    k = k + 1;
```

```
sbox[i] = Tmp;
} while (k < 0x100);
```

As always, I don't recommend reverse engineering the algorithm but instead guess that it is RC4 and then use [Cyberchef](#), Python or [Binary Refinery](#) to confirm your guess. For completeness, here is a BinRef pipeline for one of the calls:

```
> emit "36423605a96002d7e5af770baecc2d2ec1a69b7e6b1e47a95f1fbb840b96ebb5d69fe1053e7f7266bb29215d5f8ec74406561b881b2509a1t

43 Bytes, 50.29% entropy, ASCII text, with no line terminators
-----[PEEK]--
Global\206D87E0-0E60-DF25-DD8F-8E4E7D1E3BF0
-----
```

Scripting Snippets

In this section, I will share a few useful snippets when using Java to write scripts for Ghidra for malware reverse engineering. The first one is a helper function that accepts a function name, assumes there is only one function with that name and returns a list of addresses where this function is called. We will also use the `getOriginalBytes` function from a [previous blog post](#).

```
private List<Address> getCallAddresses(Function deobfuscator) {
    List<Address> addresses = new ArrayList<Address>();
    for (Reference ref : getReferencesTo(deobfuscator.getEntryPoint())) {
        if (ref.getReferenceType() != RefType.UNCONDITIONAL_CALL)
            continue;
        addresses.add(ref.getFromAddress());
    }

    return addresses;
}
```

The function `setComment` will set a comment in both the disassembly view and the decompiled view:

```
private void setComment(Address address, String comment) {
    setPlateCommentToDisassembly(address, comment);
    setCommentToDecompiledCode(address, comment);
}

private void setPlateCommentToDisassembly(Address address, String comment) {
    currentProgram.getListing().getCodeUnitAt(address).setComment(CodeUnit.PLATE_COMMENT, comment);
}

private void setCommentToDecompiledCode(Address address, String comment) {
    currentProgram.getListing().getCodeUnitAt(address).setComment(CodeUnit.PRE_COMMENT, comment);
}
```

Finally, the following function is my Q&D approach to convert a byte array to an ASCII string (even if it is a wide string). A more experienced Java developer may be able to implement it in a more beautiful way, but that's simply not me.

```
private String AsciiDammit(byte[] data, int len) {
    boolean isWide = true;
    byte[] nonWide = new byte[len / 2];
    for (int i = 0; i < len / 2; i++) {
        if (data[i * 2 + 1] != '\0') {
            isWide = false;
            break;
        }
    }
    nonWide[i] = data[i * 2];
}
return new String(isWide ? nonWide : data);
}
```

Function Arguments

As described in the "Layout" section above, we will need to get the values passed to a function call. This is a bit more involved: the `getConstantCallArgument` function below accepts a memory address of a function call and a list of integers in the variable `argumentIndices`. These integers should specify the indices of function arguments the caller wants the value of (starting with 1). The function will return an array of optional longs: it has the same length as `argumentIndices` and contains the determined value if possible.

To determine the value, `getConstantCallArgument` decompiles the function that contains the call (`caller` in the Java code), retrieves a so-called "high-level function structure" via `getHighFunction` and then uses the function `traceVarnodeValue` to retrieve the values of the requested parameters. This `traceVarnodeValue` function is an incomplete implementation of a Pcode traversal. In at least two samples, it worked though, so I still think it is worth sharing.

```
class UnknownVariableCopy extends Exception {
    public UnknownVariableCopy(PcodeOp unknownCode, Address addr) {
        super(String.format("unknown opcode %s for variable copy at %08X", unknownCode.getMnemonic(), addr.getOffset()));
    }
}

private OptionalLong[] getConstantCallArgument(Address addr, int[] argumentIndices) throws IllegalStateException, UnknownVariableCopy {
    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);
    DecompilerResults 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(index));
                argumentPos++;
            }
        }
    }
    return argumentValues;
}

private OptionalLong traceVarnodeValue(Varnode argument) throws UnknownVariableCopy {
    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:
                // known cases where an array is indexed
                return OptionalLong.empty();
            default:
                // don't know how to handle this yet.
                throw new UnknownVariableCopy(ins, argument.getAddress());
        }
    }
    return OptionalLong.of(argument.getOffset());
}
```

Automated Deobfuscation

Equipped with ways to retrieve all calls to a specific function, get values of parameters to this calls and also be able to add annotations to Ghidra, we are only missing the actual deobfuscation function. As described above in the "Decryption" section, it is RC4. Instead of doing it the enterprise way ((by calling `Cipher.getInstance("RC4");` I guess)), we will use a [random implementation by some guy on github](#). I even found myself an excuse for this: If one, at some point in the future, encounters a modified version of RC4, it is easily possible to do similar modifications in the code.

So putting all the pieces together, we end up with the following `run` method, which I will explain a bit below:

```
public void run() throws Exception {
    String deobfuscatorName;
    try {
        deobfuscatorName = askString("Enter Name", "Enter the name of the deobfuscation function below:", getFunctionBefore());
    } catch (CancelledException X) {
        return;
    }

    Function deobfuscator = getGlobalFunctions(deobfuscatorName).get(0);
    OUTER_LOOP: for (Address callAddr : getCallAddresses(deobfuscator)) {
        monitor.setMessage(String.format("parsing call at %08X", callAddr.getOffset()));

        int arguments[] = { 1, 2, 3, 4 };
        OptionalLong options[] = getConstantCallArgument(callAddr, arguments);
        for (OptionalLong option : options) {
            if (option.isEmpty()) {
                println(String.format("Argument to call at %08X is not a constant string.", callAddr.getOffset()));
                continue OUTER_LOOP;
            }
        }

        long blobAddress = options[0].getAsLong();
        int keyOffset = (int) options[1].getAsLong();
        int keyLength = (int) options[2].getAsLong();
        int dataLength = (int) options[3].getAsLong();
        if (dataLength == 0 || keyLength == 0)
            continue;

        byte[] key = getOriginalBytes(toAddr(blobAddress + keyOffset), keyLength);
        byte[] data = getOriginalBytes(toAddr(blobAddress + keyOffset + keyLength), dataLength);
        byte[] decrypted = new RC4(key).encrypt(data);

        String deobfuscated = AsciiDammit(decrypted, dataLength);
        println(String.format("%08X : %s", callAddr.getOffset(), deobfuscated));
        setComment(callAddr, String.format("Deobfuscated: %s", deobfuscated));
        createBookmark(callAddr, "DeobfuscatedString", deobfuscated);
    }
}
```

The function first asks the user for a function name. It pre-populates the field with the currently selected function or, if the script was called before, with the previous input. Even though simple, this feels surprisingly good from a user experience (UX) perspective.

The function then iterates over all calls to this deobfuscation function and retrieves values for arguments 1-4. If all of them are set, they are assigned to the following variables:

- `blobAddress` reference to the address of the blob of encrypted data in the malware
- `keyOffset` offset into that blob
- `keyLength` length of the key buffer starting at the offset into the blob
- `dataLength` length of the data buffer starting directly after the key buffer

The function then retrieves the key and the encrypted data and uses the `RC4` class to deobfuscate it. The result is then passed to the `AsciiDammit` function, which will also take care of decoding wide-strings. It then prints the address of the call and the deobfuscated string to the console, sets a comment in the disassembly and the decompile views and, creates a bookmark to the call, so we can easily look at a list of all deobfuscated strings enabling bottom-up analysis.

The [full script without explanation can be found on github](#).

Decrypted strings

For google-ability, here is a list of all deobfuscated strings in the analysed sample:

Address	Deobfuscated String
0x00401B2F	exp
0x0040151E	pk
0x00401538	pid
0x00401552	sub
0x0040156C	dbg
0x00401589	wht
0x004015A4	wfld
0x004015BF	wipe
0x004015D9	prc
0x004015F6	dmn
0x00401610	net
0x0040162B	nbody
0x00401646	nname
0x00401664	img
0x0040167B	fast
0x00401838	none
0x00401851	true
0x0040186D	false
0x004019BE	-noLan
0x00401B9C	SOFTWARE\recfg
0x00401BB5	rnd_ext
0x00401EA7	{UID}
0x00401EC0	{KEY}
0x00401ED9	{EXT}
0x00401EF5	{USERNAME}
0x00401F14	{NOTENAME}
0x00401F30	SYSTEM
0x00401F46	USER
0x00401CCF	SOFTWARE\recfg
0x00401CE8	stat
0x00401D71	{"ver":%d,"pid": "%s", "sub": "%s", "pk": "%s", "uid": "%s", "sk": "%s", "unm": "%s", "net": "%s", "grp": "%s", "lng": "%s", "bro": "%s", "dsk": "%s", "ext": "%s"}
0x00401FF4	.lock
0x004020D1	{UID}
0x004020EA	{KEY}
0x00402103	{EXT}

Address	Deobfuscated String
0x00402184	{EXT}
0x00402216	SOFTWARE\recfg
0x00402232	sub_key
0x0040224B	pk_key
0x00402264	sk_key
0x00402280	0_key
0x00403933	.bmp
0x00403E5F	cmd.exe
0x00403E7E	/c vssadmin.exe Delete Shadows /All /Quiet & bcdedit /set {default} recoveryenabled No & bcdedit /set {default} boots ignoreallfailures
0x00404085	SYSTEM\CurrentControlSet\services\Tcpip\Parameters
0x0040409E	Domain
0x004040FA	WORKGROUP
0x00404199	Control Panel\International
0x004041B2	LocaleName
0x004042A6	%08X%08X
0x0040432F	SOFTWARE\Microsoft\Windows NT\CurrentVersion
0x00404348	productName
0x004043E0	explorer.exe
0x004044B5	Globa\206D87E0-0E60-DF25-DD8F-8E4E7D1E3BF0
0x00404BF9	runas
0x004058B3	qJiQmi65SC9GfVbj
0x0040660D	\\?\A:\
0x00406547	\\?\UNC
0x00405C02	CreateStreamOnHGlobal
0x00405D3B	ole32.dll
0x00406B6A	win32kfull.sys
0x00406B83	win32k.sys
0x004012A0	fld
0x004012B7	fls
0x004012CE	ext
0x004030B7	https://
0x004030F7	wp-content
0x0040311B	static
0x00403140	content
0x00403162	include
0x00403185	uploads
0x004031A4	news

Address	Deobfuscated String
0x004031C0	data
0x004031DF	admin
0x00403264	images
0x00403287	pictures
0x004032AA	image
0x004032CC	temp
0x004032E8	tmp
0x00403308	graphic
0x00403327	assets
0x00403345	pics
0x00403360	game
0x00403441	jpg
0x00403457	png
0x00403470	gif
0x00406849	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:64.0) Gecko/20100101 Firefox/64.0
0x00406919	POST
0x0040697E	Content-Type: application/octet-stream\nConnection: close
0x004027BC	program files
0x004027D5	program files (x86)
0x0040281F	sql
0x00405C40	advapi32.dll
0x00405C79	crypt32.dll
0x00405CB2	gdi32.dll
0x00405CF6	mpr.dll
0x00405F20	shell32.dll
0x00405F59	shlwapi.dll
0x00405F92	user32.dll
0x00405FCB	winhttp.dll
0x00406004	winmm.dll

References

- [OALabs - IDA Pro Automated String Decryption For REvil Ransomware](#)
- [Python Scripting Cheat Sheet by Thomas Roth](#)