Kaspersky ICS CERT

VNC vulnerability research

Pavel Cheremushkin

22.11.2019 Version 1.0

Contents

In this article, we discuss the findings of research which covered several different implementations of a remote access system called Virtual Network Computing (VNC). As a result of this research, we identified a number of memory corruption vulnerabilities, which have been assigned a total of 37 CVE identifiers. Some of the vulnerabilities identified, if exploited, could lead to remote code execution.

Preparing for the research

The VNC system is designed to provide one device with remote access to another device's screen. It is worth noting that the protocol's specification does not limit the choice of OS and allows cross-platform implementations. There are implementations both for common operating systems – GNU/Linux, Windows, Android – and for exotic ones.

VNC has become one of the most widespread systems of its kind, thanks in part to cross-platform implementations and open-source licenses. The exact number of installations is hard to estimate. Based on data from shodan.io, over 600,000 VNC servers are available online. If you add those devices which are only available on the local network, it can be confidently assumed that the total number of VNC servers in use is many times (perhaps by orders of magnitude) greater.

According to our data, VNC is actively used in industrial automation systems. We have recently published an article on the use of remote administration tools in industrial control systems on our website. It is estimated in the article that various remote administration tools (RAT), including VNC, are installed on about 32% of industrial control system computers. In 18.6% of all cases, RATs are included in ICS software distribution packages and are installed with that software. The remaining 81.4% were apparently installed by honest or not-so-honest employees of these enterprises or their contractors. In an article published on our website, we described attacks we had analyzed, in which the attackers had installed and used remote administration tools. Importantly, in some cases the attackers exploited vulnerabilities in remote administration tools as part of their attack scenarios.

According to our estimates, most ICS vendors implement remote administration tools for their products based on VNC rather than any other system. This made an analysis of VNC security a high-priority task for us.

In 2019, the BlueKeep vulnerability (CVE-2019-0708) in Windows RDP (Remote Desktop Services) triggered an emotional public response. The vulnerability enabled an unauthorized attacker to achieve remote code execution with SYSTEM privileges on a Windows machine on which the RDP server was running. It affected 'junior' versions of the operating system, such as Windows 7 SP1 and Windows 2008 Server SP1 and SP2.

Some VNC server components in Windows are implemented as services that provide privileged access to the system, which means they themselves also have high-level access to the system. This is one more reason for prioritizing research on the security of VNC.

System description

VNC (Virtual Network Computing) is a system designed to provide remote access to the operating system's user interface (desktop). VNC uses the RFB (remote frame buffer) protocol to transfer screen images, mouse movement and keypress events between devices. As a rule, each implementation of the system includes a server component and a client component. Since the RFB protocol is standardized, different implementations of the client and server parts are interchangeable. The server component sends the image of the server's desktop to the client for viewing and the client in turn transmits client-side events (such as mouse cursor movements, keypresses, data copying and pasting via the cut buffer) back to the server. This enables the user on the client side to work on the remote machine where the VNC server is running.

The VNC server sends an image every time the remote machine's desktop is updated, which can occur, among other things, as a result of the client's actions. Sending a new complete screenshot over the network is obviously a relatively resource-intensive operation, so instead of sending the entire screenshot the protocol updates those pixels which have changed as a result of some actions or events. RFB also supports several screen update compression and encoding methods. For example, compression can be performed using zlib or RLE (run-length encoding).

Although the software is designed to perform a simple task, it has sufficiently extensive functionality for programmers to make mistakes at the development stage.

Possible attack vectors

Since the VNC system consists of server and client components, below we look at two main attack vectors:

1. An attacker is on the same network with the VNC server and attacks it to gain the ability to execute code on the server with the server's privileges.
2. A user connects to an attacker's 'server' using a VNC client and the attacker exploits vulnerabilities in the client to attack the user and execute code on the user's machine.

Attackers would without doubt prefer remote code execution on the server. However, most vulnerabilities are found in the system's client component. In part, this is because the client component includes code designed to decode data sent by the server in all sorts of formats. It is while writing data decoding components that developers often make errors resulting in memory corruption vulnerabilities.

The server part, on the other hand, can have a relatively small codebase, designed to send encoded screen updates to the user and handle events received from the client side. According to the specification, the server must support only six message types to provide all the functions required for its operation. This means that most server components have almost no complicated functionality, reducing the chances of a developer making an error. However, various extensions are implemented in some systems to augment the server's functionality, such as file transfer, chat between the client and the server, and many others. As our research demonstrated, it is in the code designed to augment the server's functionality that the majority of errors were made.

Objects of research

We selected the most common VNC implementations for our research:

- LibVNC – an open-source cross-platform library for creating a custom application based on the RFB protocol. The server component of LibVNC is used, for example, in VirtualBox to provide access to the virtual machine via VNC.
- UltraVNC – a popular open-source VNC implementation developed specifically for Windows. Recommended by many industrial automation companies for connecting to remote HMI interfaces over the RFB protocol (see, for example, here and here).
- TightVNC1.X – one more popular implementation of the RFB protocol. Recommended by many industrial automation system vendors for connecting to HMI interfaces from *nix machines.
- TurboVNC – an open-source VNC implementation. Uses the libjpeg-turbo library to compress JPEG images in order to accelerate image transfer.

As part of our research, we did not analyze the security of a very popular product called RealVNC, because the product's license does not allow reverse engineering.

Prior research

Before beginning to analyze VNC implementations, it is essential to do reconnaissance and see what vulnerabilities have already been identified in each of them.

In 2014, the Google Security Team published a small LibVNC vulnerability analysis report. Since the project includes a very small amount of code, it could be assumed that Google engineers had identified all vulnerabilities existing in LibVNC. However, I was able to find several issues on GitHub (for example, this and this), which were created later than 2014.

The number of vulnerabilities identified in the UltraVNC project is not large. Most of these vulnerabilities have to do with the exploitation of a simple stack overflow with arbitrary length data being written to a fixed-size buffer on the stack.

All known vulnerabilities were found a relatively long time ago. Since then, project codebase has grown, while the older codebase was found to include old vulnerabilities.

Research findings

LibVNC

After analyzing previously identified vulnerabilities, I fairly easily found variants of some of these vulnerabilities in the code of the extension providing file transfer functionality. The extension is not enabled by default: developers must explicitly allow it to be used in their LibVNC based projects. This is probably why these vulnerabilities had not been identified before.

Next, I moved on from analyzing server code to researching the client part. It was there that I found vulnerabilities which had the most critical importance for the project and which were also quite diverse.

Among the vulnerabilities identified, it is worth mentioning several classes of vulnerabilities that will also come up in other projects based on the RFB protocol.

It can be said that each of these classes was made possible by the way the protocol's specification is designed. More precisely, the protocol's specification is designed in a way that does not guard developers against these classes of bugs, enabling such flaws to appear in the code.

As an illustration of this point, you can look at the structures used in VNC projects to handle network messages. For example, open the rfbproto.h file, which has been used by generations of VNC project developers since 1999. The file is included in the LibVNC project, among others.

An excellent example for demonstrating the first class of vulnerabilities is the *rfbClientCutTextMsg* structure, which is used to send information on cut buffer changes on the client to the server.

```
typedef struct {
    uint8_t type; /* always rfbClientCutText */
    uint8_t pad1;
    uint16_t pad2;
    uint32_t length;
    /* followed by char text[length] */
} rfbClientCutTextMsg;
```

After establishing a connection and performing an initial handshake, during which the client and the server agree to use specific screen settings, all messages transferred have the same format. Each message starts with one byte, which represents the message type. Depending on message type, a message handler and structure matching the type are selected. In different VNC clients, the structure is filled in more or less in the same way (pseudocode in C):

```
ReadFullData(socket, ((char *)&msg) + 1, sz_rfbServerSomeMessageType - 1);
```

In this way, the entire message structure is filled in, with the exception of the first byte, which defines the message type. It can be seen that all fields in the structure are controlled by the remote user. It should also be noted that *msg* is a *union*, which consists of all possible message structures.

Since the contents of the cut buffer has an unspecified length, memory will be allocated for it dynamically, using `malloc`. It should also be remembered that the cut buffer field should presumably contain text and it is customary to terminate text data with the zero character in the C language. Given all this, as well as the fact that the field *length* has the type uint32_t and is fully controlled by the remote user, in this case we have a typical integer overflow (pseudocode in C):

```
char *text = malloc(msg.length + 1);
ReadFullData(socket, text, msg.length);
text[msg.length] = 0;
```

If an attacker sends a message length field with a value equal to UINT32_MAX = 2^{32}- 1 = 0xffffffff, the function `malloc(0)` will be called as a result of an integer overflow. If the standard glibc malloc memory allocation mechanism is used, the call will return a chunk of the smallest possible size – 16 bytes. At the same time, a length equal to UINT32_MAX will be passed to the `ReadFullData` function as an argument, which, in the case of LibVNC, will result in a heap-based buffer overflow.

The second vulnerability type can be demonstrated on the same structure. As one can read in the specification or the RFC, some structures include padding for field alignment. However, from a security researcher's viewpoint, this is just one more opportunity to discover a memory initialization error (see here and here). Let's have a look at this typical error (pseudocode in C):

```
rfbClientCutTextMsg cct;
cct.type = rfbClientCutText;
cct.length = length;
WriteToRFBServer(socket, &cct, sz_rfbClientCutTextMsg);
WriteToRFBServer(socket, str, len);
```

The message structure is created on the stack, after which **some** of its fields are filled in and the structure is sent to the server. It can be seen that the structures *pad1* and *pad2* remain empty. As a result of this, an uninitialized variable is sent over the network and an attacker can read uninitialized memory from the stack. If the attacker is in luck, the memory area that the attacker is able to access may contain the address of the heap, stack or text section, enabling the attacker to bypass ASLR and use overflow to achieve remote code execution on the client.

Such trivial vulnerabilities have been found in VNC projects sufficiently often, which is why we decided to place them into separate classes.

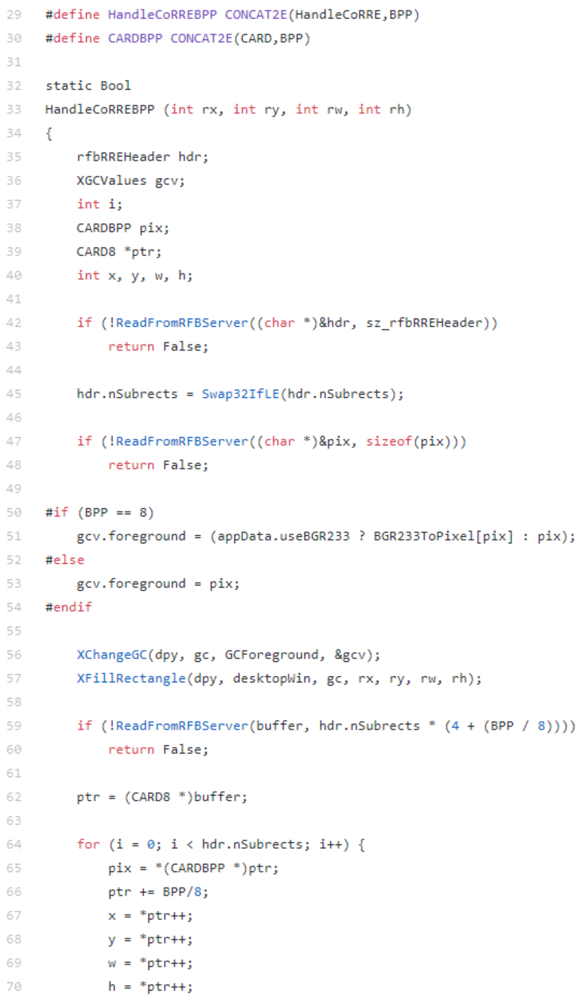
It is worth noting that analyzing such projects as LibVNC, which are positioned as cross-platform solutions, is not an easy task. While doing research on such projects, one should ignore anything that has to do with the specific OS and architecture of the researcher's computer and view the project exclusively through the prism of the C language standard, otherwise it's easy to miss some obvious flaws in code, which can only be reproduced on a specific platform. For example, in this case, the heap overflow vulnerability was incorrectly fixed on the 32 bit platform because the size or the *size_t* type on the x86_64 platform is different from the same type's size on the 32 bit x86 platform.

Information on all vulnerabilities identified was provided to developers and the vulnerabilities were closed (some even twice, thanks to Solar Designer for the help).

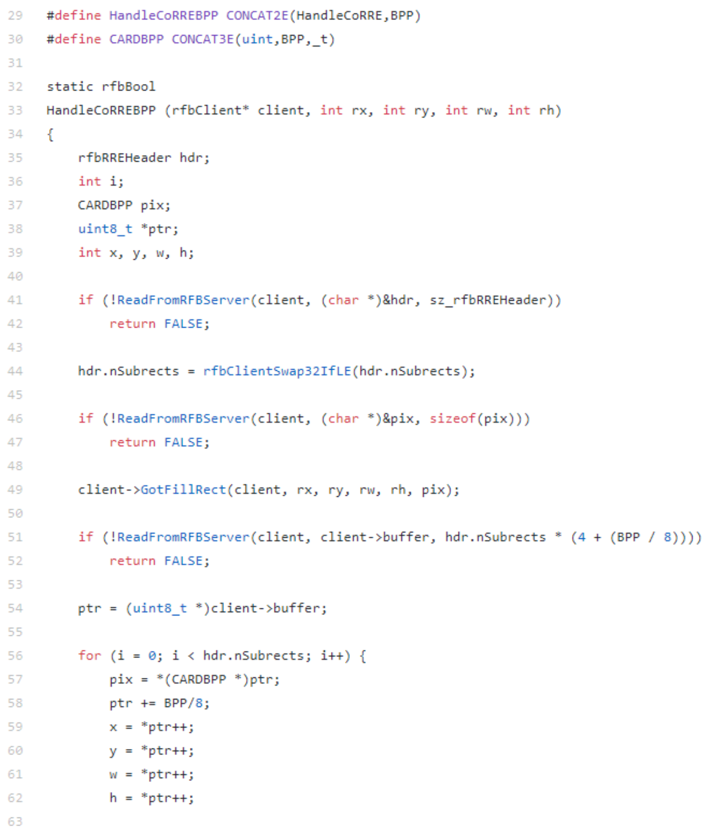
TightVNC

The next target for research was a fairly popular VNC client implementation for GNU/Linux.

I was able to identify vulnerabilities in that system very quickly, because most were fairly straightforward and some were identical to those found in LibVNC. Two code fragments from two different projects are compared below.



```c
#define HandleCoRREBPP CONCAT2E(HandleCoRRE,BPP)
#define CARDBPP CONCAT2E(CARD,BPP)

static Bool
HandleCoRREBPP (int rx, int ry, int rw, int rh)
{
    rfbRREHeader hdr;
    XGCValues gcv;
    int i;
    CARDBPP pix;
    CARD8 *ptr;
    int x, y, w, h;

    if (!ReadFromRFBServer((char *)&hdr, sz_rfbRREHeader))
        return False;

    hdr.nSubrects = Swap32IfLE(hdr.nSubrects);

    if (!ReadFromRFBServer((char *)&pix, sizeof(pix)))
        return False;

#if (BPP == 8)
    gcv.foreground = (appData.useBGR233 ? BGR233ToPixel[pix] : pix);
#else
    gcv.foreground = pix;
#endif

    XChangeGC(dpy, gc, GCForeground, &gcv);
    XFillRectangle(dpy, desktopWin, gc, rx, ry, rw, rh);

    if (!ReadFromRFBServer(buffer, hdr.nSubrects * (4 + (BPP / 8))))
        return False;

    ptr = (CARD8 *)buffer;

    for (i = 0; i < hdr.nSubrects; i++) {
        pix = *(CARDBPP *)ptr;
        ptr += BPP/8;
        x = *ptr++;
        y = *ptr++;
        w = *ptr++;
        h = *ptr++;
```



```c
#define HandleCoRREBPP CONCAT2E(HandleCoRRE,BPP)
#define CARDBPP CONCAT3E(uint,BPP,_t)

static rfbBool
HandleCoRREBPP (rfbClient* client, int rx, int ry, int rw, int rh)
{
    rfbRREHeader hdr;
    int i;
    CARDBPP pix;
    uint8_t *ptr;
    int x, y, w, h;

    if (!ReadFromRFBServer(client, (char *)&hdr, sz_rfbRREHeader))
        return FALSE;

    hdr.nSubrects = rfbClientSwap32IfLE(hdr.nSubrects);

    if (!ReadFromRFBServer(client, (char *)&pix, sizeof(pix)))
        return FALSE;

    client->GotFillRect(client, rx, ry, rw, rh, pix);

    if (!ReadFromRFBServer(client, client->buffer, hdr.nSubrects * (4 + (BPP / 8))))
        return FALSE;

    ptr = (uint8_t *)client->buffer;

    for (i = 0; i < hdr.nSubrects; i++) {
        pix = *(CARDBPP *)ptr;
        ptr += BPP/8;
        x = *ptr++;
        y = *ptr++;
        w = *ptr++;
        h = *ptr++;

```

Originally, this vulnerability was identified in the LibVNC project, in the CoRRE decoding method (see code on the right-hand side). In the above code fragment, data of arbitrary length is read to a fixed-length buffer inside the *rfbClient* structure. This naturally results in buffer overflow. By a curious coincidence, function pointers are located inside the structure, almost right after the cut buffer, which almost immediately results in code execution.

It can be observed that, with the exception of some minor variations, the code fragments from LibVNC and TightVNC can be considered identical. Both fragments were copied from the AT&T Laboratories. The developers introduced this vulnerability back in 1999. (I was able to determine this through the AT&T Laboratories license, in which developers usually specify who was involved in the development project during different time periods.) That code has been modified several times since then – for example, in LibVNC the static global buffer was moved to the client's structure – but the vulnerability survived all the modifications.

It is also worth noting that *HandleCoRREBPP* is a rather original name. If you search the code of projects on GitHub for this character combination, you can find lots of VNC-related projects that thoughtlessly copied the vulnerable decoding function carrying this name or the entire LibVNC library. This is why these projects may remain vulnerable forever – unless the developers update the contents of their projects or fix the vulnerability in the code themselves.

The character combination *HandleCoRREBPP* is in fact not a function name. BPP in this case stands for "Bits per Pixel" and is a number equal to 8, 16 or 32, depending on the color depth agreed on by the client and the server at the initialization stage. It is assumed that developers will use this file as an auxiliary file in their macros as follows:

```
#ifndef HandleCoRRE8
#define BPP 32
#include ”corre.h”
#undef BPP
#endif
```

The result is several functions: *HandleCoRRE8, HandleCoRRE16*, and *HandleCoRRE32*.

Since the program was originally written in C rather than C++, the developers had to come up with such tricks because there were no templates available. However, if you google the function name *HandleCoRRE* or *HandleCoRRE32*, you may discover that there are projects which were slightly modified, either using or not using patterns, but which still contain the vulnerability. Unfortunately, there are hundreds of projects in which this code was included without any changes or copied and it is not always possible to contact their developers.

The sad story of TightVNC does not end here. When we reported the vulnerabilities to TightVNC developers, they thanked us for the information and let us know that they had discontinued the development of the TightVNC 1.X line and no longer fixed any vulnerabilities found, because it had become uneconomical for their company. At some point, GlavSoft began to develop a new line, TightVNC 2.X, which does not include any GPL-licensed third-party code and which can therefore be developed as a commercial product. It should be noted that TightVNC 2.X for Unix systems is distributed only under commercial licenses and should not be expected to be released as open source software.

We reported the vulnerabilities identified in TightVNC oss-security and emphasized that package maintainers needed to fix these vulnerabilities by themselves. Although we sent our notification to package maintainers in January 2019, the vulnerabilities had not been fixed at the time of this article’s publication (November 2019).

TurboVNC

This VNC project deserves a special ‘prize’: the one vulnerability identified in it is mind-boggling.

Consider a C code fragment taken from the main server function designed to handle user messages:

```
char data[64];
READ(((char *)&msg) + 1, sz_rfbFenceMsg - 1)
READ(data, msg.f.length)
if (msg.f.length > sizeof(data))
    rfbLog("Ignoring fence.  Payload of %d bytes is too large.\n",
           msg.f.length);
else
    HandleFence(cl, flags, msg.f.length, data);
return;
```

This code fragment reads a message in the *rfbFenceType* format. The message provides the server with information on the length *msg.f.length* of type *uint8_t* user data, which follows the message. This is obviously the case of arbitrary user data being read into a fixed-size buffer, resulting in stack overflow. Importantly, a check of the length of the data read is performed **after** the data has been read into the buffer.

Due to the absence of overflow protection on the stack (a so-called canary), this vulnerability makes it possible to control return addresses and, consequently, to achieve remote code execution on the server. An attacker would, however, first need to obtain authentication credentials to connect to the VNC server or gain control of the client before the connection is established.

UltraVNC

In UltraVNC, I was able to identify multiple vulnerabilities in both the server and the client components of the project, to which 22 CVE IDs were assigned.

A distinguishing feature of this project is its focus on Windows systems. When analyzing projects that can be compiled for GNU/Linux, I prefer to take two different approaches to vulnerability search. First, I analyze the code, looking for vulnerabilities in it. Second, I try to figure out how the search for vulnerabilities in the project can be automated using fuzzing. This is what I did when analyzing LibVNC, TurboVNC, and TightVNC. For such projects, it is very easy to write a wrapper for libfuzzer, since the project does not depend on a specific operating system's implementation of the network API – there is an additional abstraction layer implemented for that. To write a good fuzzer, all you have to do is implement the target function on your own, as well as rewrite the networking functions. This will allow data from the fuzzer to be fed to the program – as if it was transferred over the network.

However, in the case of analyzing projects for Windows, the latter technique is difficult to use even with open-source projects because the relevant tools are either not available or poorly developed. At the time of the analysis, libfuzzer for Windows had not yet been released. In addition, the event-oriented approach used in Windows application development means that a very large amount of code would have to be rewritten to achieve good fuzzing coverage. Because of this, I used only manual code analysis when analyzing UltraVNC for vulnerabilities.

As a result of this analysis, I found an entire 'zoo' of vulnerabilities in UltraVNC – from trivial buffer overflows in *strcpy* and *sprintf* to more or less curious vulnerabilities that can rarely be encountered in real-world projects. Below we discuss some of these vulnerabilities.

CVE-2018-15361

This vulnerability exists in UltraVNC client-side code. At the initialization stage, the server should provide information on display height and width, color depth, palette and name of the desktop, which can be displayed, for example, in the title bar of the window.

The name of the desktop is a string of an undefined length. Consequently, the string's length is sent to the client first, followed by the string itself. The relevant fragment of code is shown below:

```
void ClientConnection::ReadServerInit()
{
    ReadExact((char *)&m_si, sz_rfbServerInitMsg);

    m_si.framebufferWidth = Swap16IfLE(m_si.framebufferWidth);
    m_si.framebufferHeight = Swap16IfLE(m_si.framebufferHeight);
    m_si.format.redMax = Swap16IfLE(m_si.format.redMax);
    m_si.format.greenMax = Swap16IfLE(m_si.format.greenMax);
    m_si.format.blueMax = Swap16IfLE(m_si.format.blueMax);
    m_si.nameLength = Swap32IfLE(m_si.nameLength);

    m_desktopName = new TCHAR[m_si.nameLength + 4 + 256];
    m_desktopName_viewonly = new TCHAR[m_si.nameLength + 4 + 256+16];
    ReadString(m_desktopName, m_si.nameLength);
. . .
}
```

The attentive reader will make the correct observation that the above code contains an integer overflow vulnerability. However, in this case the vulnerability leads not to heap-based buffer overflow in the `ReadString` function but to more curious consequences.

```
void ClientConnection::ReadString(char *buf, int length)
{
    if (length > 0)
        ReadExact(buf, length);
    buf[length] = '\0';
}
```

It can be seen that the *ReadString* function is designed to read a string of the length *length* and terminate it with a zero. It is worth noting that the function takes the signed type as its second argument.

If we specify a very large number in *m_si.nameLength*, it will be treated as a negative number when passed to the *ReadString* function as an argument. This will result in *length* failing the positivity check and the *buf* array remaining unitialized. Only one thing that will happen: a null byte will be written at offset *buf* + *length*. Given that *length* is a negative number, this makes it possible to write the null byte at a fixed negative offset relative to *buf*.

The upshot of this is that if an integer overflow occurs when allocating *m_desktopName* and the buffer is allocated on the regular heap of the process, this will make it possible to write the null byte to the previous chunk. If an integer overflow does not occur and the system has sufficient memory, a large buffer will be allocated, with a new heap allocated for it. With the right parameters, a remote attacker would be able to write a null byte to the *_NT_HEAP* structure, which will be located directly before a huge chunk. This vulnerability is guaranteed to cause a DoS, but the question of the ability to achieve remote code execution remains open. I wouldn't rule out that experts in exploiting the Windows userland heap could turn this vulnerability into an RCE if they wanted to.

CVE-2019-8262

The vulnerability was identified in the handler of data encoded using the Ultra encoding. It demonstrates that the security and availability of this functionality really hung by a very thin thread.

The handler uses the *lzo1x_decompress* function from the minilzo library. To understand what the vulnerability is, one has to look at the prototypes of compression and decompression functions.

To call the decompression function, one has to pass the buffer containing compressed data, compressed data length, the buffer to which the data should be unpacked and its length as inputs. It should be kept in mind that the function may return an error if the input data cannot be decompressed. In addition, the developer needs to know the exact length of the data that will be unpacked to the output buffer. This means that, in addition to the error code, the function should return a value equal to the number of bytes written. For example, the argument that is used to pass the write buffer length can be used for this, provided that it is passed by pointer. In that case the minimum interface of the decompression function will look as follows:

```
int decompress(const unsigned char *in, size_t in_len, unsigned char *out, size_t
*out_len)
```

The first four parameters of this function are the same as the first four parameters of the *lzo1x_decompress* function.

Now consider the fragment of UltraVNC code that contains the critical heap overflow vulnerability.

```
void ClientConnection::ReadUltraRect(rfbFramebufferUpdateRectHeader *pfburh) {

    UINT numpixels = pfburh->r.w * pfburh->r.h;

    UINT numRawBytes = numpixels * m_minPixelBytes;
    UINT numCompBytes;
    lzo_uint new_len;
    rfbZlibHeader hdr;

    // Read in the rfbZlibHeader
    omni_mutex_lock l(m_bitmapdcMutex);
    ReadExact((char *)&hdr, sz_rfbZlibHeader);
    numCompBytes = Swap32IfLE(hdr.nBytes);

    CheckBufferSize(numCompBytes);
    ReadExact(m_netbuf, numCompBytes);
    CheckZlibBufferSize(numRawBytes);

lzo1x_decompress((BYTE*)m_netbuf,numCompBytes,(BYTE*)m_zlibbuf,&new_len,NULL);
       . . .
}
```

As you can see, UltraVNC developers do not check the `lzo1x_decompress` return code, which is, however, insignificant compared to another flaw – the improper use of *new_len*.

The uninitialized variable *new_len* is passed to the `lzo1x_decompress` function. At the time of calling the function, the variable should be equal to the length of the *m_zlibbuf* buffer. In addition, while debugging vncviewer.exe (the executable file was taken from a build on the UltraVNC official website), I was able to find out why this code had passed the testing stage. It turned out that the problem was that, since the variable *new_len* was not initialized, it contained a large text section address value. This made it possible for a remote user to pass specially crafted data to the decompression function as inputs to ensure that the function, when writing to the *m_zlibbuf* buffer, would write the data beyond the buffer's boundary, resulting in heap overflow.

Conclusion

In conclusion, I would like to mention that while doing the research I often couldn't help thinking that the vulnerabilities I found were too unsophisticated to have been missed by everyone before. However, it was true. Each of these vulnerabilities had a very long lifetime.

Some of the vulnerability classes identified in the study are present in a large number of open-source projects, surviving even codebase refactoring. I believe it is very important to be able to systematically identify such sets of vulnerable projects containing vulnerabilities that are not always inherited in clear ways.

Almost none of the projects analyzed are unit tested; programs are not systematically tested for security using static code analysis or fuzzing. Magic constants that are abundant in code make it similar to a house of cards: just one constant changed in this unstable structure could result in a new vulnerability.

Here are our recommendations for developers and vendors that use third-party VNC project code in their products:

- Set up a bug tracking mechanism in all third-party VNC projects used and regularly update their code to the latest release.
- Add compilation options that make it harder for attackers to exploit any vulnerabilities that may exist in the code. Even if researchers are not able to identify all the vulnerabilities in a project, exploiting them should be made as difficult as possible.

  For example, some of the vulnerabilities described in this article would be impossible to exploit to achieve remote code execution if the project was compiled as a position-independent executable (PIE). In that case, the vulnerabilities would remain, but their exploitation would lead to denial of service (DoS) rather than RCE.

  Another example is the unfortunate experience with TurboVNC: the compiler can sometimes optimize the procedure of checking the stack canary. Some compilers perform such optimizations by removing stack canary checks from the functions that don't have explicitly allocated arrays. However, the compiler could make a mistake and fail to check for the presence of a buffer in some of the structures on the stack or in switch-case statements (which is what probably happened in the case of TurboVNC). To make it impossible to exploit a vulnerability that has been identified, the compiler should be explicitly told that the stack canary checking procedure should not be optimized.
- Perform fuzzing and testing of the project on all architectures for which the project is made available. Some vulnerabilities may manifest themselves only on one of the platforms due to its specific features.
- Be sure to use sanitizers in the process of fuzzing and at the testing stage. For example, a memory sanitizer is guaranteed to identify such vulnerabilities as the use if uninitialized values.

On the positive side, password authentication is often required to exploit server-side vulnerabilities, and the server may not allow users to configure a password-free authentication method for security reasons. This is the case, for example, with UltraVNC. As a safeguard against attacks, clients should not connect to unknown VNC servers and administrators should configure authentication on the server using a unique strong password.

The following CVE were registered based on this research:

1. LibVNC

CVE-2018-6307
CVE-2018-15126
CVE-2018-15127
CVE-2018-20019
CVE-2018-20020
CVE-2018-20021
CVE-2018-20022
CVE-2018-20023
CVE-2018-20024
CVE-2019-15681

2. TightVNC

CVE-2019-8287
CVE-2019-15678
CVE-2019-15679
CVE-2019-15680

3. TurboVNC

CVE-2019-15683

4. UltraVNC

CVE-2018-15361
CVE-2019-8258
CVE-2019-8259
CVE-2019-8260
CVE-2019-8261
CVE-2019-8262
CVE-2019-8263
CVE-2019-8264
CVE-2019-8265
CVE-2019-8266
CVE-2019-8267
CVE-2019-8268
CVE-2019-8269
CVE-2019-8270
CVE-2019-8271
CVE-2019-8272
CVE-2019-8273
CVE-2019-8274
CVE-2019-8275
CVE-2019-8276
CVE-2019-8277
CVE-2019-8280

To be continued…

Kaspersky Industrial Control Systems Cyber Emergency Response Team (Kaspersky ICS CERT) is a global project of Kaspersky aimed at coordinating the efforts of automation system vendors, industrial facility owners and operators, and IT security researchers to protect industrial enterprises from cyberattacks. Kaspersky ICS CERT devotes its efforts primarily to identifying potential and existing threats that target industrial automation systems and the industrial internet of things.

Kaspersky ICS CERT ics-cert@kaspersky.com