Kaspersky ICS CERT

kaspersky

Исследование уязвимостей в VNC

Павел Черемушкин

13.11.2019 Version 1.0

Оглавление

В этой статье мы рассказываем о результатах исследования различных реализаций системы удаленного доступа Virtual Network Computing (VNC). Результатами данного исследования являются обнаруженные нами уязвимости порчи памяти, для которых в сумме было выделено 37 идентификаторов CVE. Эксплуатация некоторых обнаруженных уязвимостей приводит к возможности удалённого исполнения кода.

Подготовка к исследованию

Система VNC предназначена для предоставления одному устройству удалённого доступа к экрану другого. При этом спецификация протокола не ограничивает выбор ОС и позволяет кроссплатформенные реализации. Существуют реализации как для распространенных ОС – GNU/Linux, Windows, Android, – так и для экзотических.

На сегодняшний день система VNC, в том числе благодаря возможности кроссплатформенных реализаций и лицензии с открытым кодом, стала одной из самых распространённых. Точное количество инсталляций оценить сложно. Согласно данным с shodan.io, количество VNC-серверов, доступных из глобальной сети, составляет более 600000. С учётом устройств, доступных только внутри локальной сети, можно с уверенностью сказать, что общее количество используемых VNC-серверов многократно (возможно, на порядки) больше.

По нашим данным, VNC активно используется на объектах промышленной автоматизации. Не так давно на нашем сайте вышла статья об угрозах при использовании средств удалённого администрирования в системах промышленной автоматизации. По оценкам, приведённым в статье, около 32% компьютеров в индустриальной сети имеют различные установленные средства для удалённого администрирования, в том числе VNC. 18,6% из них включены в состав инсталляционных пакетов ПО АСУ ТП и устанавливаются вместе с ним. Остальные 81,4% инсталлированы, очевидно, добросовестными или не очень добросовестными сотрудниками самих предприятий и подрядных организаций. В одной из наших статей мы описали исследованные нами атаки, когда подобные средства устанавливались и использовались злоумышленниками. При этом в ряде случаев злоумышленники использовали уязвимости средств удалённого администрирования в сценариях своих атак.

По нашей оценке, большинство вендоров систем АСУ ТП реализуют средства удалённого администрирования своими продуктами именно на основе VNC. Это сделало для нас исследование безопасности VNC одним из приоритетных.

В 2019 году бурную реакцию общественности вызвала уязвимость BlueKeep CVE-2019-0708 в службе удалённого рабочего стола Windows RDP (Remote Desktop Services). Она позволяла без авторизации получить удалённое исполнение кода с правами SYSTEM на машине под управлением ОС Windows, на которой запущен RDP server. Уязвимость касалась «младших» версий этой операционной системы, например, Windows 7 SP1 или Windows 2008 Server SP1 и SP2.

Некоторые серверные VNC компоненты на операционной системе Windows оформлены как сервисы для привилегированного доступа к системе, а значит, тоже имеют высокий уровень доступа к системе, что ещё раз показывает, насколько актуальной является задача исследования VNC.

Описание системы

VNC (Virtual Network Computing) – система для предоставления удалённого доступа к пользовательскому интерфейсу (рабочему столу) ОС. VNC использует протокол RFB (remote frame buffer) для передачи между устройствами изображения экрана, событий движения мыши и нажатия клавиш на клавиатуре. Каждая реализация данной системы, как правило, включает в себя серверную и клиентскую компоненты. За счёт того, что протокол RFB стандартизирован, различные реализации клиентской и серверной части

взаимозаменяемы. Серверная часть передаёт изображение экрана рабочего стола сервера для просмотра клиенту, а клиент, в свою очередь, транслирует события, происходящие на клиентской стороне (например, перемещение курсора мыши, нажатие клавиш, копирование и вставка данных через буфер обмена) обратно на сервер. Это позволяет пользователю на стороне клиента работать на удалённой машине, где запущен VNC-сервер.

Изображение передается VNC-сервером при каждом обновлении рабочего стола удаленной машины, которые могут происходить в том числе в результате действий клиента. Очевидно, что передавать по сети новое изображение экрана целиком – довольно дорогостоящая операция, поэтому в протоколе передаётся не всё изображение, а лишь обновление тех пикселей экрана, которые изменились в результате каких-то действий или событий. Протокол RFB также поддерживает несколько методов сжатия и кодировки этих обновлений экрана. Например, сжатие может осуществляться при помощи zlib или кодирования методом RLE.

Несмотря на простую задачу, которую должно выполнять данное программное обеспечение, в нем достаточно функциональности, чтобы на этапе разработки у программистов была возможность допустить ошибки.

Возможные векторы атак

Система VNC состоит из сервера и клиента, поэтому далее будем рассматривать два основных вектора атаки:

1. Злоумышленник находится с VNC сервером в одной сети и атакует его, чтобы получить возможность исполнения кода на сервере с его правами.
2. Пользователь при помощи VNC клиента присоединяется к «серверу» злоумышленника, который использует уязвимости в клиенте, чтобы атаковать пользователя и получить возможность исполнения кода на его машине.

Атакующие, несомненно, предпочли бы получить возможность удалённого исполнения кода на сервере. Однако большинство уязвимостей содержатся именно в клиентской части системы. Отчасти это обусловлено тем, что в клиентской части содержится код, который должен уметь декодировать данные, присылаемые сервером в самых разных форматах. Именно при написании компонентов для декодирования данных разработчики нередко допускают ошибки, которые приводят к возникновению уязвимостей порчи памяти.

Серверная часть, в свою очередь, может обойтись достаточно небольшой кодовой базой, которая отправляет пользователю закодированные обновления экрана и обрабатывает присланные с клиентской стороны события. Согласно спецификации, сервер должен поддерживать только шесть типов сообщений для обеспечения всех необходимых для работы функций. Из-за этого в большинстве серверных компонент почти нет сложной функциональности. Следовательно, уменьшается вероятность того, что разработчик сможет допустить ошибку. Однако для расширения возможностей сервера в некоторых системах имплементированы различные дополнения, такие как, например, передача файлов, чат между клиентом и сервером и многое другое. Как показало исследование, именно в этих расширениях возможностей сервера допущено наибольшее количество ошибок.

Объекты исследования

Для проведения исследования мы выбрали наиболее часто встречающиеся реализации VNC:

- LibVNC – кроссплатформенная библиотека с открытым исходным кодом для создания собственного приложения на базе протокола RFB. Серверный компонент LibVNC используется, например, в VirtualBox для предоставления доступа через VNC к виртуальной машине.
- UltraVNC – популярная реализация VNC с открытым исходным кодом. Разработана исключительно для OS Windows. Рекомендована к использованию многими компаниями промышленной автоматизации для подключения по протоколу RFB к удалённому интерфейсу HMI (см., например, здесь и здесь).
- TightVNC1.X – также является популярной реализацией протокола RFB. Рекомендована многими производителями систем промышленной автоматизации для подключения к интерфейсу HMI с *nix машин.
- TurboVNC – реализация VNC с открытым исходным кодом. Использует библиотеку libjpeg-turbo для сжатия изображения в формате JPEG с целью ускорения передачи изображения.

В рамках нашего исследования мы не исследовали безопасность очень популярного продукта RealVNC, так как лицензия продукта не позволяет выполнять обратную инженерию.

Предшествующие исследования

Прежде чем приступить к исследованию реализаций VNC, необходимо произвести разведку и посмотреть на то, какие уязвимости уже были найдены в каждой из них.

В 2014 году Google Security Team опубликовала небольшой отчёт об исследовании LibVNC на предмет уязвимостей. Поскольку в проекте очень мало кода, можно было предположить, что инженеры компании Google нашли все уязвимости в LibVNC. Однако мне удалось найти несколько тикетов на GitHub (например, этот и этот), которые появились позднее 2014 года.

В проекте UltraVNC уязвимостей было найдено не много. Большинство из них ограничиваются эксплуатацией простого переполнения стека с данными произвольной длины, которые записываются в буфер на стеке фиксированного размера.

Все известные уязвимости были найдены довольно давно. С того времени кодовая база проектов увеличилась, а в старой кодовой базе, как выяснилось, сохранились давние уязвимости.

Результаты исследования

LibVNC

После анализа уже найденных ранее уязвимостей мне удалось довольно легко найти вариации таких же уязвимостей в коде расширения для передачи файлов. Данное расширение не включено по умолчанию: разработчику необходимо явно разрешить его использование в своём проекте на базе LibVNC. Вероятно, именно поэтому уязвимости не были найдены ранее.

Далее я перешёл от исследования серверного кода к клиентской части. Именно в ней я обнаружил наиболее критичные для данного проекта уязвимости, которые были ещё и довольно разнообразными.

Среди обнаруженных уязвимостей хочется отметить несколько классов, которые будут встречаться и в других проектах на базе протокола RFB.

Можно сказать, что каждый из данных классов стал возможен благодаря особенностям спецификации протокола. Точнее, спецификация протокола такова, что не ограждает разработчиков от ошибок этих классов, а позволяет их совершать.

Достаточно взглянуть на структуры, которые используются для обработки сетевых сообщений в VNC проектах. Откроем файл *rfbproto.h*, который используется поколениями разработчиков VNC проектов, начиная с 1999 года. Этот файл содержится и в проекте LibVNC.

Прекрасным примером для демонстрации первого класса уязвимостей может служить структура *rfbClientCutTextMsg*, которая используется для передачи серверу информации об изменении содержимого буфера обмена на клиенте.

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

После установки соединения и начального «рукопожатия», в ходе которого клиент и сервер соглашаются использовать определённые настройки экрана, все передаваемые сообщения имеют единый формат. Каждое из них начинается с одного байта, который обозначает тип сообщения. В зависимости от типа выбирается обработчик сообщения и соответствующая типу структура, которая в разных клиентах VNC заполняется примерно одинаковым способом (псевдокод на языке Си):

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

Таким образом заполняется вся структура сообщения, кроме первого байта, который определяет тип сообщения. Как можно видеть, все поля структуры контролируются удалённым пользователем. Стоит также отметить, что *msg* – это *union*, который состоит из всех возможных структур сообщений.

Так как содержимое буфера имеет неопределённую длину, то память для него будет выделена динамически, с помощью `malloc`. Необходимо также помнить, что поле

буфера обмена предположительно должно быть текстовым, а в языке программирования Си текстовые данные принято терминировать нулём. Учитывая все эти факты, а также то, что поле *length* имеет тип uint32_t и полностью контролируется удалённым пользователем, в данном случае мы имеем типичное переполнение целочисленного типа (псевдокод на языке Си):

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

Если атакующий передаёт длину сообщения, равную UINT32_MAX = 2^{32}- 1 = 0xffffffff, то в результате переполнения целочисленного типа будет вызвана функция *malloc*(0). В случае если используется стандартный механизм аллокации памяти glibc malloc, такой вызов вернёт чанк (chunk) самого маленького размера – 16 байт. При этом в функцию *ReadFullData* в качестве аргумента будет передана длина, равная UINT32_MAX, что в случае LibVNC приводит к переполнению буфера на куче.

Второй класс уязвимостей можно продемонстрировать на этой же структуре. Как можно прочитать в спецификации или RFC, некоторые структуры включают в себя паддинг для выравнивания полей. Однако с точки зрения исследователя безопасности это лишь ещё одна возможность обнаружить ошибку инициализации памяти (см. здесь и здесь). Рассмотрим данную типичную ошибку (псевдокод на языке Си):

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

Структура сообщения создаётся на стеке, после чего заполняются **некоторые** её поля, и структура отправляется на сервер. Как можно видеть, поля структуры *pad1* и *pad2* остаются незаполненными. Это приводит к тому, что неинициализированные данные отправляются по сети, и атакующий может прочитать неинициализированную память со стека. При определённом везении там может находиться адрес кучи, стека или текстовой секции, что позволит атакующему обойти ASLR и при помощи переполнения получить удалённое исполнение кода на клиенте.

Такие банальные уязвимости встречались в VNC проектах достаточно часто, поэтому мы и решили выделить их в отдельные классы.

Стоит отметить, что исследование таких проектов как LibVNC, которые позиционируют себя как кроссплатформенные решения, является непростой задачей. В ходе их исследования следует абстрагироваться от особенностей ОС и архитектуры компьютера исследователя и рассматривать проект через призму стандарта языка Си, иначе можно пропустить некоторые очевидные ошибки в коде, которые воспроизводятся только на определённой платформе. Например, в данном случае в силу того, что размер типа *size_t* на x86_64 отличается от размера данного типа на 32-битной платформе x86, уязвимость переполнения кучи была некорректно закрыта на 32-битной платформе.

Информация обо всех обнаруженных уязвимостях была передана разработчикам, и уязвимости были закрыты (некоторые даже дважды, спасибо Solar Designer за помощь).

TightVNC

Следующей целью для исследования стала довольно популярная реализация клиента VNC для GNU/Linux.

Уязвимости в данной системе удалось обнаружить очень быстро, потому что большинство из них были довольно простые, а некоторые и вовсе были такими же, как и в LibVNC. Ниже приведено сравнение двух фрагментов кода из двух разных проектов.

```
#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++;
```

```
#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++;
```

Данная уязвимость изначально была обнаружена в проекте LibVNC в методе декодирования CoRRE (см. код справа). В данном фрагменте кода происходит считывание данных произвольной длины в буфер фиксированной длины внутри структуры *rfbClient*. Естественно, это приводит к переполнению буфера. По случайности внутри данной структуры, практически вслед за буфером, располагаются указатели на функции, что почти моментально приводит к исполнению кода.

Как можно видеть, за исключением небольших изменений фрагменты кодов из LibVNC и TightVNC можно считать одинаковыми. Эти фрагменты скопированы у AT&T Laboratories. Разработчики привнесли данную уязвимость ещё в 1999 году. (Мне удалось это выяснить при помощи лицензии AT&T Laboratories, в которой разработчики обычно указывают, кто и в какой временной промежуток занимался разработкой проекта.) С тех пор этот код был несколько раз модифицирован – например, в LibVNC

статический глобальный буфер был перенесён в структуру клиента, – однако уязвимость смогла пережить все правки.

Также стоит сказать, что *HandleCoRREBPP* является довольно оригинальным именем. Если поискать данную комбинацию символов в коде проектов на GitHub, то можно обнаружить большое количество проектов, связанных с VNC, которые бездумно скопировали уязвимую функцию декодирования с этим именем или скопировали полностью библиотеку LibVNC. Именно поэтому данные проекты могут навсегда остаться уязвимыми – если их разработчики не обновят содержимое своих проектов или не исправят уязвимость в коде самостоятельно.

Хотя на самом деле набор символов *HandleCoRREBPP* не является названием функции. BPP в данном случае означает “Bits Per Pixel” и является числом 8, 16 или 32 в зависимости от того, на какую глубину цвета договорились клиент и сервер на этапе инициализации. Подразумевается, что в другом файле разработчик будет использовать данный файл как вспомогательный в своих макросах следующим образом:

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

В результате получится несколько функций *HandleCoRRE8, HandleCoRRE16* и *HandleCoRRE32*.

Так как программа изначально была написана на языке Си, а не Си++, то из-за отсутствия шаблонов разработчикам пришлось прибегать к подобным ходам. Однако, если погуглить имя функции *HandleCoRRE* или *HandleCoRRE32*, то можно обнаружить, что есть проекты, которые были немного переписаны с использованием шаблонов или без, но, несмотря на это, уязвимость в них сохранилась. К сожалению, количество проектов, которые скопировали или дословно переписали этот код, исчисляется сотнями, и связаться с их разработчиками не всегда представляется возможным.

На этом грустная часть истории с TightVNC не заканчивается. После того, как мы сообщили разработчикам TightVNC об уязвимостях в их продукте, они поблагодарили нас за информацию и сказали, что развивать линейку TightVNC 1.X и исправлять в ней уязвимости они более не намерены, так как это не приносит их компании экономической выгоды. С определенного момента компания GlavSoft начала развивать новую линейку TightVNC 2.X, которая не содержит стороннего кода под лицензией GPL, соответственно, её можно развивать как коммерческий продукт. Следует упомянуть, что TightVNC 2.X для Unix систем распространяется только под коммерческими лицензиями, и выхода этого ПО в open source ждать не стоит.

Мы сообщили в oss-security и мейнтейнерам пакетов о найденных уязвимостях в TightVNC и о необходимости их самостоятельно исправить. Несмотря на наше уведомление мейнтейнерам, отправленное в январе 2019 года, ко времени публикации статьи (ноябрь 2019) уязвимости исправлены не были.

TurboVNC

Данный VNC проект заслуживает отдельной “премии” в рамках этой статьи, так как в нём была обнаружена всего одна уязвимость, но её существование, безусловно, поражает воображение.

Рассмотрим отрезок кода на Си, взятый из главной функции обработки пользовательских сообщений сервером:

```
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;
```

В данном фрагменте кода происходит считывание сообщения формата *rfbFenceType*. В этом сообщении сервер узнаёт длину пользовательских данных *msg.f.length* типа *uint8_t*, которые следуют за этим сообщением. Очевидно, что в данном случае происходит считывание произвольных пользовательских данных в буфер фиксированного размера, что приводит к переполнению на стеке. При этом проверка длины считываемых данных производится **после** того, как данные считываются в буфер.

В связи с отсутствием на стеке защиты от переполнения (так называемой «канарейки») данная уязвимость даёт возможность контролировать адреса возврата и, соответственно, возможность удалённого исполнения кода на сервере. Злоумышленнику, правда, сначала понадобится получить данные аутентификации для подключения к серверу VNC или получить контроль над клиентом до момента установки соединения.

UltraVNC

В UltraVNC удалось обнаружить множество уязвимостей, которые содержатся в серверной и клиентской компонентах проекта, для которых было выделено 22 идентификатора CVE.

Одной из особенностей этого проекта является его специализация на Windows системах. При исследовании проектов, которые можно скомпилировать под GNU/Linux, я предпочитаю подходить к поиску уязвимостей с двух сторон. Во-первых, я анализирую код с целью найти в нём уязвимость. Во-вторых, пытаюсь понять, как можно автоматизировать поиск уязвимостей в этом проекте при помощи фаззинга. Именно так я поступил в ходе анализа LibVNC, TurboVNC и TightVNC. Для таких проектов очень легко написать обёртку для libfuzzer, так как проект не зависит от реализации сетевого API конкретной ОС – для этого реализован дополнительный уровень абстракции. Для написания хорошего фаззера достаточно имплементировать самостоятельно таргет-функцию, а также переписать функции по работе с сетью. Таким образом на вход программе будут подаваться данные из фаззера – как будто они были переданы по сети.

Однако, в случае анализа проектов для Windows из-за отсутствия или неразвитости соответствующего инструментария такая техника плохо применима даже для проектов с открытым исходным кодом. libfuzzer для Windows во время проведения исследований еще не вышел. Также из-за использования событийно-ориентированного подхода при разработке приложений под Windows, чтобы добиться хорошего покрытия при фаззинге пришлось бы переписать очень большое количество кода. Поэтому для поиска уязвимостей в UltraVNC я использовал исключительно ручной анализ кода

В итоге, в UltraVNC удалось найти целый «зоопарк» самых разных уязвимостей – от банальных переполнений буфера в *strcpy* и *sprintf* до более-менее интересных уязвимостей, которые редко можно встретить на практике. Рассмотрим некоторые из них.

CVE-2018-15361

Данная уязвимость содержится в клиентском коде UltraVNC. На этапе инициализации сервер должен сообщить высоту и ширину дисплея, глубину цвета, палитру и имя рабочего стола, которое можно отобразить, например, в заголовке окна.

Имя рабочего стола является строкой неопределённой длины, следовательно, клиенту сначала передаётся длина, а потом и сама строка. Ниже приведён описанный фрагмент кода.

```
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);
. . .
}
```

Внимательный читатель скажет, что в данном коде содержится уязвимость integer overflow, и будет прав. Но в данном случае эта уязвимость приводит не к переполнению буфера на куче в функции *ReadString*, а к более интересным последствиям.

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

Как можно видеть, функция *ReadString* должна считывать строку длины *length* и терминировать её нулём. Стоит обратить внимание на то, что функция принимает в качестве второго аргумента знаковый тип.

В случае если мы зададим очень большое число в *m_si.nameLength*, то, когда оно попадёт в функцию *ReadString*, оно будет восприниматься как отрицательное число. Это приведёт к тому, что *length* не пройдёт проверку на положительность, и массив *buf* останется неинициализированным. Единственное, что произойдёт -- по смещению *buf* + *length* будет записан нуль-байт. Учитывая, что *length* является отрицательным числом, это даёт возможность записать нулевой байт по фиксированному отрицательному смещению по отношению к *buf*.

Таким образом, если при аллокации *m_desktopName* произойдёт целочисленное переполнение, и данный буфер будет аллоцирован на обычной куче процесса, то это даст возможность записать нуль-байт в предыдущий чанк. Если целочисленного переполнения не произойдёт, и у системы будет достаточно памяти, то будет выделен большой буфер, для которого будет аллоцирована новая куча. При подходящих параметрах у удалённого атакующего появится возможность записать нуль-байт в структуру *_NT_HEAP*, которая будет находиться прямо перед гигантским чанком. Данная уязвимость гарантированно приводит к DoS, но вопрос о возможности получения удалённого исполнения кода остаётся открытым. Я не исключаю, что эксперты в области эксплуатации userland кучи операционной системы Windows смогли бы при желании довести данную уязвимость до RCE.

CVE-2019-8262

Уязвимость была обнаружена в обработчике данных в кодировке Ultra. Она показывает, на сколь «тонком волоске» держалась безопасность и работоспособность данной функциональности этого ПО.

Обработчик использует функцию *lzo1x_decompress* из библиотеки minilzo. Для того чтобы понять, в чём заключается уязвимость, необходимо посмотреть на прототипы функций сжатия и распаковки.

Для вызова функции декомпрессии необходимо передать на вход буфер со сжатыми данными, длину сжатых данных, буфер, в который необходимо распаковать данные, и его длину. Нужно учитывать, что функция может вернуть ошибку, если данные, поступившие на вход, невозможно разжать. Кроме того, разработчику необходимо знать точную длину данных, которые будут распакованы в выходной буфер. То есть, помимо кода ошибки функция также должна возвращать значение количества записанных байт. Для этого, например, можно использовать тот же самый аргумент, что и для передачи длины буфера записи, если передать его по указателю. Тогда минимальный интерфейс функции декомпрессии будет выглядеть так:

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

Первые четыре параметра этой функции совпадают с первыми четырьмя параметрами функции *lzo1x_decompress*.

Теперь рассмотрим фрагмент кода UltraVNC, содержащий критическую уязвимость переполнения кучи.

```
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);
       . . .
}
```

Как можно видеть, разработчики UltraVNC не проверяют код возврата *lzo1x_decompress*, что, впрочем, является несущественным недостатком по сравнению с другой ошибкой – неправильным использованием *new_len*.

В функцию *lzo1x_decompress* передаётся неинициализированная переменная *new_len*, которая на момент вызова функции должна быть равна длине буфера *m_zlibbuf.* Кроме того, при отладке vncviewer.exe (исполняемый файл был взят из сборки с официального сайта UltraVNC) мне удалось выяснить, почему данный код прошёл этап тестирования. Проблема оказалась в том, что в *new_len*, вследствие того, что данная переменная не была инициализирована, содержалось большое значение адреса текстовой секции. Это позволяет удалённому пользователю передать на вход такие данные, что функция распаковки при записи в буфер *m_zlibbuf* выйдет за его границы, что приведет к переполнению на куче.

Заключение

В заключение хочется сказать, что в ходе проведения данного исследования я неоднократно ловил себя на мысли о том, что обнаруженные уязвимости слишком простые, чтобы их никто не заметил до меня. Однако это оказалось реальностью. Время жизни каждой уязвимости было очень долгим.

Некоторые классы уязвимостей, обнаруженные в результате исследования, содержатся в большом количестве open-source проектов и сохраняются в них даже после рефакторинга кодовой базы. Я считаю, что очень важно на систематической основе уметь обнаруживать такие множества проектов, содержащих иногда неявно унаследованные уязвимости.

Почти во всех проанализированных проектах отсутствуют юнит-тесты, не проводится систематическое тестирование программ на безопасность при помощи статического анализа кода или фаззинга. За счёт того, что код зачастую наполнен магическими константами, его можно сравнить с карточным домиком: в этой неустойчивой конструкции изменение одной константы может привести к появлению уязвимости.

Позитивной стороной является то, что для эксплуатации серверных уязвимостей зачастую необходима парольная аутентификация, при этом из соображений безопасности сервер может не позволять пользователю установить беспарольный метод аутентификации. Именно таким образом это, например, реализовано в UltraVNC. Для того чтобы обезопасить себя от злоумышленников, клиентам не стоит подключаться к неизвестным VNC серверам, а администраторам надо настроить аутентификацию на сервере с сильным уникальным паролем.

Разработчикам и производителям, которые используют в своих продуктах код сторонних VNC проектов, рекомендуем:

- Настроить механизм отслеживания багов во всех используемых сторонних VNC проектах и регулярно обновлять их код до последнего релиза.
- Добавить опции компиляции, усложняющие эксплуатацию возможных уязвимостей для атакующего. Даже если исследователям не удастся обнаружить все существующие в проекте уязвимости, можно максимально усложнить процесс их эксплуатации.

  Например, некоторые из описанных в статье уязвимостей невозможно было бы проэксплуатировать с целью получения удалённого исполнения кода, если бы проект был скомпилирован с динамической базой исполняемого файла (PIE). В таком случае уязвимость всё равно существовала бы, но результатом её эксплуатации был бы отказ в обслуживании (DoS), а не RCE.

  Другим примером может служить печальный опыт TurboVNC: компилятор может оптимизировать процедуру проверки стековой канарейки. Некоторые компиляторы осуществляют данную оптимизацию путём удаления проверки стековой канарейки из функций, в которых нет явно аллоцированых массивов. Однако компилятор может ошибиться и не проверить наличие буфера внутри какой-нибудь из структур на стеке или внутри switch-case выражений (как это вероятно произошло в случае с TurboVNC). Для того чтобы сделать невозможной эксплуатацию обнаруженной уязвимости, необходимо явно указывать компилятору, что нельзя оптимизировать процедуру проверки канарейки на стеке.

- Проводить фаззинг и тестирование проекта на всех архитектурах, для которых поставляется продукт. Некоторые уязвимости могут проявлять себя только на одной платформе в силу её особенностей.
- Обязательно использовать санитайзеры в ходе фаззинга и на этапе тестирования. Например, с помощью memory sanitizer можно гарантированно обнаружить такие уязвимости как использование неинициализированного значения.

Позитивной стороной является то, что для эксплуатации серверных уязвимостей зачастую необходима парольная аутентификация, при этом из соображений безопасности сервер может не позволять пользователю установить беспарольный метод аутентификации. Именно таким образом это, например, реализовано в UltraVNC. Для того чтобы обезопасить себя от злоумышленников, клиентам не стоит подключаться к неизвестным VNC серверам, а администраторам надо настроить аутентификацию на сервере с сильным уникальным паролем.

По результатам исследования были зарегистрированы следующие CVE:

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

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

Продолжение следует….

Центр реагирования на инциденты информационной безопасности промышленных инфраструктур «Лаборатории Касперского» (Kaspersky ICS CERT) — глобальный проект «Лаборатории Касперского», нацеленный на координацию действий производителей систем автоматизации, владельцев и операторов промышленных объектов, исследователей информационной безопасности при решении задач защиты промышленных предприятий и объектов критически важных инфраструктур.

Kaspersky ICS CERT ics-cert@kaspersky.com