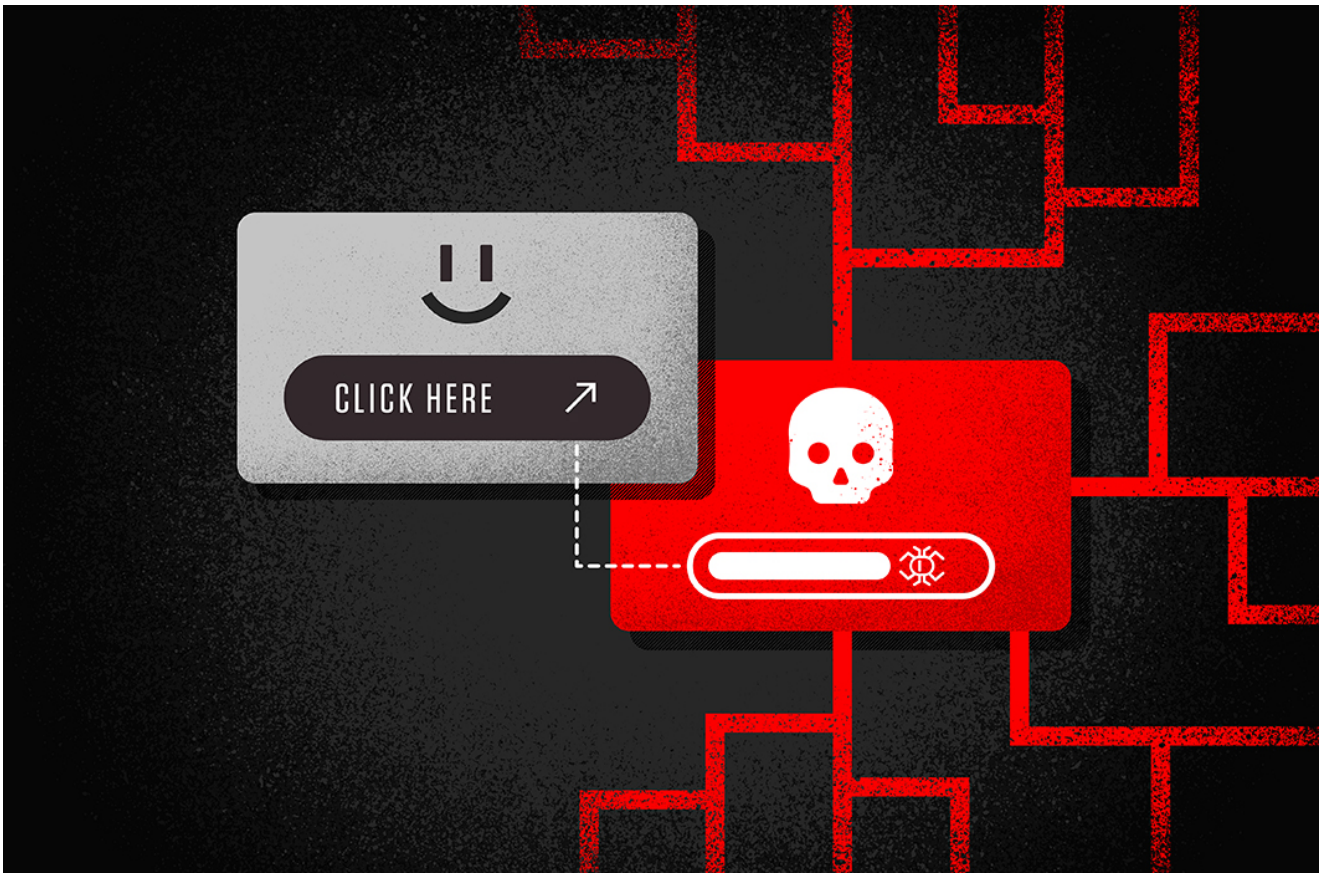
One-Click Attack Surface in Linux Desktop Environments

crowdstrike.com/blog/one-click-attack-surface-in-linux-desktop-environments/

Lukas Kupczyk - Max Julian Hofmann November 9, 2021



The Advanced Research Team at CrowdStrike Intelligence discovered multiple vulnerabilities affecting libvncclient. In some widely used desktop environments, such as GNOME, these vulnerabilities can be triggered in a one-click fashion.

Introduction

Client-side exploitation has become a crucial component of many attackers' toolkits. In the desktop space, exploiting browsers is considered to be one of the most impactful capabilities, but due to continuous hardening measures and wide adoption of sandboxing, it is also one of the most complex. However, other components of a typical desktop environment have not been subject to the same scrutiny and can therefore pose risks that go unnoticed. Sparked by our own observations of applications helpfully spawning applications at the click of a link, we decided to investigate the security posture of a typical Linux desktop environment.

URL Scheme Handlers

Whether the desktop starts an application when the user clicks on a link (e.g., in an email or an instant message) depends on whether a handler application is registered for a URL scheme. In a stock Ubuntu 21.04 desktop based on Gnome, there are various applications registered as handlers for specific URL schemes. A full list of these handlers can be retrieved by grepping the `mimeinfo.cache` file as follows:

```
$ grep x-scheme-handler /usr/share/applications/mimeinfo.cache
x-scheme-handler/apt=apturl.desktop;
x-scheme-handler/chrome=firefox.desktop;
x-scheme-handler/ftp=firefox.desktop;
x-scheme-handler/ghelp=yelp.desktop;
x-scheme-handler/help=yelp.desktop;
x-scheme-handler/http=firefox.desktop;
x-scheme-handler/https=firefox.desktop;
x-scheme-handler/icy=org.gnome.Totem.desktop;
x-scheme-handler/icyx=org.gnome.Totem.desktop;
x-scheme-handler/info=yelp.desktop;
x-scheme-handler/magnet=transmission-gtk.desktop;
x-scheme-handler/mailto=thunderbird.desktop;
x-scheme-handler/man=yelp.desktop;
x-scheme-handler/mms=org.gnome.Totem.desktop;
x-scheme-handler/mmsh=org.gnome.Totem.desktop;
x-scheme-handler/net=org.gnome.Totem.desktop;
x-scheme-handler/pnm=org.gnome.Totem.desktop;
x-scheme-handler/rdp=org.remmina.Remmina.desktop;remmina-file.desktop;
x-scheme-handler/remmina=org.remmina.Remmina.desktop;remmina-file.desktop;
x-scheme-handler/rtmp=org.gnome.Totem.desktop;
x-scheme-handler/rtp=org.gnome.Totem.desktop;
x-scheme-handler/rtsp=org.gnome.Totem.desktop;
x-scheme-handler/snap=snap-handle-link.desktop;
x-scheme-handler/spice=org.remmina.Remmina.desktop;remmina-file.desktop;
x-scheme-handler/uvox=org.gnome.Totem.desktop;
x-scheme-handler/vnc=org.remmina.Remmina.desktop;remmina-file.desktop;
x-scheme-handler/vnd.libreoffice.cmis=libreoffice-startcenter.desktop;
```

While a URL starting with a custom scheme might seem suspicious to cautious users, the exact target of a link can be hidden in applications that allow embedding HTML markup. An example of such an application is the personal information management application Evolution:

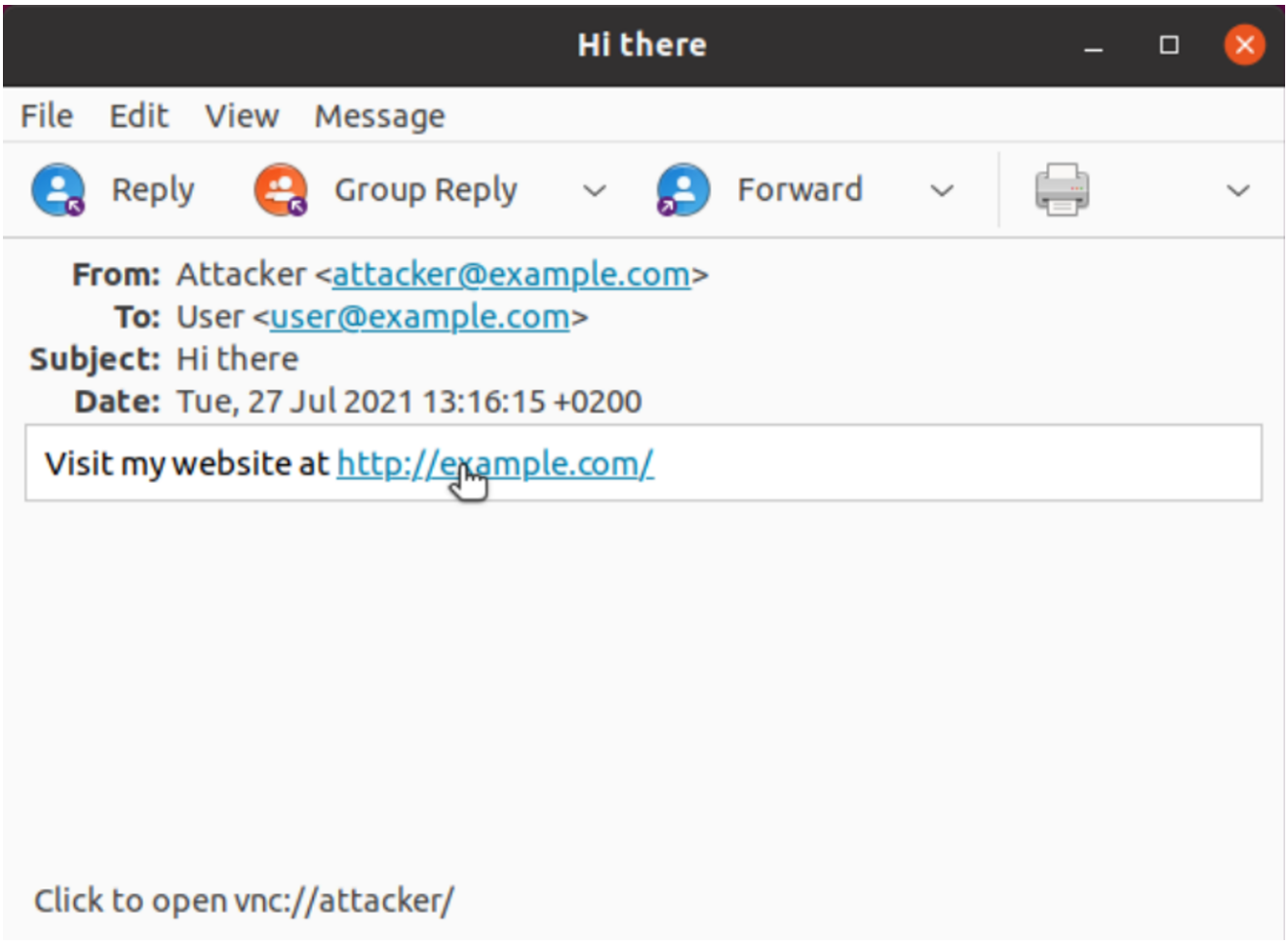


Figure 1. Example of how a link target can be hidden in the Evolution application

In the previous URL handler listing, we have observed that the remote administration tool Remmina is registered for the URL schemes `rdp`, `remmina`, `spice` and `vnc`. The corresponding `.desktop` file specified as part of the handler definition contains a reference to the shell script `remmina-file-wrapper`, which receives the specified URL as its second argument (`%U` placeholder):

```
$ cat /usr/share/applications/remmina-file.desktop
[Desktop Entry]
[...]
Exec=remmina-file-wrapper -c %U
Icon=org.remmina.Remmina
MimeType=application/x-remmina;x-scheme-handler/remmina;x-scheme-handler/rdp;x-
scheme-handler/spice;x-scheme-handler/vnc;
[...]
```

That shell script then starts the actual Remmina binary in case a matching URL is specified:

```
$ cat /usr/bin/remmina-file-wrapper
#!/usr/bin/env bash
[...]

REMMINA="/usr/bin/remmina"

if [[ ! -f "$REMMINA" ]] ; then
    REMMINA="${USRBIN}/remmina"
else
    REMMINA="remmina"
fi

export GLADE_HOME="$USRBIN/../share/remmina/ui/"

case "$@" in
    *rdp:*)
        "$REMMINA" "${@#rdp:\/\/}"
        ;;
    *spice:*)
        "$REMMINA" "${@#spice:\/\/}"
        ;;
    *vnc:*)
        "$REMMINA" "${@#vnc:\/\/}"
        ;;
    *remmina:*)
        "$REMMINA" "${@#remmina:\/\/}"
        ;;
    *)
        "$REMMINA" "${@}"
        ;;
esac
```

These URL handlers therefore open up quite a large attack surface that is reachable with minimal user interaction. As an example, clicking on a URL such as `vnc://user:pass@example.com/` automatically starts Remmina and instantly establishes a VNC connection with the given credentials to the specified remote host. Any vulnerability affecting Remmina's VNC protocol implementation would therefore potentially allow for one-click exploitation scenarios. We therefore decided to take a closer look at libvncclient, which is used by Remmina for its VNC support.

Code Auditing

Libvncclient implements the VNC protocol, which is based on the concept of a remote frame buffer. It offers functionality to transfer keystrokes and mouse inputs from the client to the server while relaying the server's graphical desktop as framebuffer updates to the client. The original specification of VNC's underlying remote framebuffer protocol (RFB) was released in 1998 and was relatively simple. However, over the years, various implementations added further features to optimize different aspects of the protocol, many of them aimed at reducing

the amount of data that needs to be transferred for framebuffer updates. On the flip side, these optimizations naturally also resulted in more complexity and therefore an increased attack surface.

Two of these optimizations originate from the UltraVNC and TightVNC protocol implementations and use custom encoding schemes. For compatibility reasons, both of these schemes are also supported by libvncclient. As outlined in the following sections, a manual code audit uncovered two distinct memory corruption vulnerabilities in the handling of these encoding schemes.

Heap Buffer Overflow in Ultra Encoding

Messages from the VNC server to the client are handled by the `HandleRFBServerMessage()` function. That function first reads the message type and then continues to process the rest of the message accordingly. Framebuffer updates are transmitted in messages of type `rfbFramebufferUpdate`. These messages will then contain information about the number of rectangular framebuffer updates contained in the message, and for each rectangle, the data encoding (e.g., `rfbEncodingUltra` for UltraVNC encoding). The following source code excerpt from libvncclient shows the function `HandleUltraBPP()`, which implements the processing of UltraVNC-encoded framebuffer updates:

```
static rfbBool
HandleUltraBPP (rfbClient* client, int rx, int ry, int rw, int rh)
{
  rfbZlibHeader hdr;
  int toRead=0;
[...]
  lzo_uint uncompressedBytes = (( rw * rh ) * ( BPP / 8 ));

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

  toRead = rfbClientSwap32IfLE(hdr.nBytes);
[...]
```

The string `BPP` in the function name will be expanded by the C preprocessor into three different functions for handling frame buffer updates with 8, 16 or 32 bits of color information per pixel. As shown in the listing, after reading the header from the socket via `ReadFromRFBServer()`, its field `nBytes` is used to set the *signed* integer variable `toRead`. Afterward, `toRead` is used as the size to allocate a 4-byte aligned buffer (`ultra_buffer`) in case no previous one has been allocated in a prior invocation of the function, or if the previous one has an insufficient size:

```
[...]
  /* allocate enough space to store the incoming compressed packet */
  if ( client->ultra_buffer_size < toRead ) { if ( client->ultra_buffer != NULL ) {
      free( client->ultra_buffer );
    }
    client->ultra_buffer_size = toRead;
    /* buffer needs to be aligned on 4-byte boundaries */
    if ((client->ultra_buffer_size % 4)!=0)
      client->ultra_buffer_size += (4-(client->ultra_buffer_size % 4));
    client->ultra_buffer = (char*) malloc( client->ultra_buffer_size );
[...]
```

Afterward, the code reads the amount of data specified by `toRead` from the socket:

```
[...]
 /* Fill the buffer, obtaining data from the server. */
  if (!ReadFromRFBServer(client, client->ultra_buffer, toRead))
      return FALSE;
[...]
```

A malicious VNC server is able to fully control the value of the `toRead` variable, including negative values. As a result, `ultra_buffer` can be overflowed by first sending a frame buffer update that leads to an allocation of a certain size, followed by a second update that sets `toRead` to a negative value. Then, the buffer allocated during the first frame buffer update can be overflowed by the second one, as the condition `client->ultra_buffer_size < toRead` will not be fulfilled, leading to a call to `ReadFromRFBServer()` where `toRead` is negative.

As `ReadFromRFBServer()` expects an unsigned integer value, the implicit type conversion ultimately leads to a `read()` call with an overly large `count` value. The exact amount of data that is overwritten can be controlled by closing the underlying socket connection early.

Intra-Struct/Heap Overflow in Tight Encoding

In case Tight encoding is used for framebuffer updates, the function `HandleTightBPP()` is invoked (similar to Ultra encoding, `BPP` is a placeholder for the preprocessor). As shown in the following listing, the function reads the value `comp_ctl` from the underlying socket:

```
static rfbBool
HandleTightBPP(rfbClient *client, int rx, int ry, int rw, int rh)
{
  CARDBPP fill_colour;
  uint8_t comp_ctl;
  uint8_t filter_id;
  filterPtrBPP filterFn;
  z_streamp zs;
  int err, stream_id, compressedLen, bitsPixel;
  int bufferSize, rowSize, numRows, portionLen, rowsProcessed, extraBytes;
  rfbBool readUncompressed = FALSE;

  if (client->frameBuffer == NULL)
    return FALSE;

  if (rx + rw > client->width || ry + rh > client->height)
  {
    rfbClientLog("Rect out of bounds: %dx%d at (%d, %d)\n", rx, ry, rw, rh);
    return FALSE;
  }

  if (!ReadFromRFBServer(client, (char *)&comp_ctl, 1))
    return FALSE;
[...]
```

That value is used by the function to make a number of assumptions regarding the exact format — e.g., it is used to select a filter function, defines the type of rectangle, and signals whether compression is used. Later, the function `ReadCompactLen()` is invoked to read the value of `compressedLen` from the underlying socket:

```
[...]
  /* Read the length (1..3 bytes) of compressed data following. */
  compressedLen = (int)ReadCompactLen(client)
[...]
```

`ReadCompactLen()` allows the server to specify a length value that uses up to three bytes. The most significant bits of the first one or two bytes specify whether another byte follows. Therefore, the specified size can consist of up to 22 bits (7 + 7 + 8). The maximum integer value that can be returned by the function is therefore 4,194,303 (2 ^ 22 – 1).

In case the client expects uncompressed data (indicated by `comp_ctl`), `compressedLen` is passed to `ReadFromRFBServer()` as the amount of data that is read into `client->buffer`:

```
[...]
 if (readUncompressed)
  {
    if (!ReadFromRFBServer(client, (char *)client->buffer, compressedLen))
      return FALSE;
[...]
```

As shown below, `client->buffer` is a fixed-size array (307,200 bytes) located in the central `rfbClient` struct, which is heap-allocated for each connection:

```
typedef struct _rfbClient {
        uint8_t* frameBuffer;
[...]
        int serverPort; /**< if -1, then use file recorded by vncrec */ rfbBool
listenSpecified; int listenPort, flashPort; struct { int x, y, w, h; } updateRect;
/** Note that the CoRRE encoding uses this buffer and assumes it is big enough to
hold 255 * 255 * 32 bits -> 260100 bytes.  640*480 = 307200 bytes.
           Hextile also assumes it is big enough to hold 16 * 16 * 32 bits.
           Tight encoding assumes BUFFER_SIZE is at least 16384 bytes. */

#define RFB_BUFFER_SIZE (640*480)
        char buffer[RFB_BUFFER_SIZE];
        char *bufoutptr;
        unsigned int buffered;
[...]
```

As the maximum size of `compressedLen` can exceed the size of the `buffer` struct member, the server is able to overflow said buffer and fully control the subsequent members of the `rfbClient` struct as well as adjacent heap memory. This includes pointers through which an arbitrary write primitive can be achieved as well as function pointers that allow hijacking of the control flow.

Fuzzing

In addition to manually auditing some of the libvncclient code base, we also made use of afl++ to conduct fuzzing tests. To do so, we developed a thin wrapper around libvncclient's main message dispatching function `HandleRFBServerMessage()` . The argument for this function is a client object that is configured to read framebuffer updates from `stdin` . An introduction to the general methodology we applied to fuzzing in this case will be presented in a subsequent blog post.

Through fuzzing, we uncovered two additional bugs that are outlined below.

Heap Buffer Overflow in TRLE

In case a TRLE-encoded framebuffer update is received, the function `HandleTRLE()` is invoked. As shown in the following source code excerpt, the function contains a loop that receives an arbitrary amount of `0xFF` bytes from the underlying socket into the heap-allocated `client->raw_buffer` :

```
static rfbBool HandleTRLE(rfbClient *client, int rx, int ry, int rw, int rh) {
[...]
uint8_t *buffer;
[...]
buffer = (uint8_t*)(client->raw_buffer);
[...]
while (*buffer == 0xff) {
if (!ReadFromRFBServer(client, (char*)buffer + 1, 1))
return FALSE;
length += *buffer;
buffer++;
}
[...]
}
```

As there is no bounds check present, the buffer can be overflown by a malicious server.

Stack Buffer Overflow in ZRLE Encoding

In case a ZRLE-encoded framebuffer update is received, the function `HandleZRLE()` is invoked. As shown in the following source code excerpt, the function first reads a structure of type `rfbZRLEHeader` from the server. Afterward, additional zlib-compressed data is received.

```
static rfbBool HandleZRLE (rfbClient* client, int rx, int ry, int rw, int rh)
{

while (( remaining > 0 ) && ( inflateResult == Z_OK )) {
[...]

                /* Fill the buffer, obtaining data from the server. */
                if (!ReadFromRFBServer(client, client->buffer,toRead))
                        return FALSE;

                client->decompStream.next_in  = ( Bytef * )client->buffer;
                client->decompStream.avail_in = toRead;

                /* Need to uncompress buffer full. */
                inflateResult = inflate( &client->decompStream, Z_SYNC_FLUSH );
[...]
        }

        if ( inflateResult == Z_OK ) {
[...]
                for(j=0; j<rh; j+=rfbZRLETileHeight)
                        for(i=0; i<rw; i+=rfbZRLETileWidth) {
[...]
int result=HandleZRLETile(
client,
(uint8_t *)buf,
remaining,
rx+i,
ry+j,
subWidth,
subHeight
);
[...]
}
```

The decompressed additional data is then passed to the function `HandleZRLETILE()` as the parameter `buf`. Its first byte determines the `type` of tile that is to be processed:

```
static int HandleZRLETile(rfbClient* client, uint8_t* buffer,size_t buffer_length,
int x,int y,int w,int h)
{
[...]
uint8_t type;
[...]
        type = *buffer;
        buffer++;
[...]
        if( type == 0 ) /* raw *
[...]
        else if( type == 1 ) /* solid */
[...]
else if( type <= 127 ) { /* packed Palette */
                CARDBPP palette[16];
                int i,j,shift,
[...]
                /* read palette */
                for(i=0; i<type; i++,buffer+=REALBPP/8)
                        palette[i] = UncompressCPixel(buffer);
[...]
}
```

As shown in the above source code excerpt, in case `type` is neither 0 nor 1 but less than or equal to 127, a branch that handles a packed palette is taken. In this case, a stack-allocated array of 16 elements named `palette` is declared. The exact value of `type` then determines how many elements are read into the `palette` array inside a for-loop. In case `type`'s value exceeds 16, an out-of-bounds write on the stack occurs.

Exploitability

While the outlined vulnerabilities allow to achieve an arbitrary write primitive and to fully control the instruction pointer, the presence of address space layout randomization (ASLR) makes exploitation difficult in a default Ubuntu x64 desktop environment. This is due to the fact that Remmina itself is compiled as a position-independent executable and is therefore similar to libraries loaded at a randomized location. Thus, an attacker has no fixed addresses to rely on. However, in case one of the vulnerabilities can be turned into an information leak, or by finding a separate bug allowing to disclose memory addresses, the outlined vulnerabilities would immediately allow for remote code execution.

Conclusion

While Linux distributions have begun to provide application-specific isolation mechanisms and have also started to integrate sandboxing techniques in critical components that expose complex attack surfaces, many other parts of typical desktop environments have not been hardened in a similar fashion and remain comparably easy targets for exploitation. In the case presented here, we were able to identify several memory corruption bugs that can be triggered with minimal user interaction due to vulnerable URL handlers.

Some of these were also identified through fuzzing, and we'll document our general methodology in a subsequent blog post. The research is by far not exhaustive and should only be considered to serve as an example of the extensive attack surface exposed by custom URL handlers.

It is worth noting that Linux isn't the only platform that exposes this type of attack surface. In fact, the recently disclosed MSHTML remote code execution vulnerability (CVE-2021-40444 — for more information, see the September 2021 Patch Tuesday update) on Windows operating systems was also primarily enabled by the platform supporting and automatically adding custom protocol handlers for known file types.

Finally, we would like to thank Christian Beier of the libvnc project for working with us and developing patches for the bugs.

Additional Resources

- *Find out how CrowdStrike Intelligence uses fuzzing to hunt for bugs.*
- *Learn how CrowdStrike Falcon X combines automated analysis with human intelligence, enabling security teams, regardless of size or skill, to get ahead of the attacker's next move.*
- *Falcon X Premium adds threat intelligence reporting and research from CrowdStrike experts — enabling you to get ahead of nation-state, eCrime and hacktivist attacks.*
- *Falcon X Elite expands your team with access to an intelligence analyst to help defend against threats targeting your organization.*
- *Learn how to stop adversaries targeting your industry — schedule a free 1:1 intel briefing with a CrowdStrike threat intelligence expert today.*