Zoom In: Emulating 'Exploit Purchase' in Simulated Targeted Attacks

contextis.com/en/blog/zoom-in-simulated-targeted-attacks

1. Home
2. Blog
3. Zoom In: Emulating 'Exploit Purchase' in Simulated Targeted Attacks

Zoom In: Emulating 'Exploit Purchase' in Simulated Targeted Attacks

Context regularly perform Red Team and Simulated Attacker engagements for several clients. These simulated attacks aim to uplift our client's ability to respond to real word adversaries. We perform these assessments by firstly identifying actionable intelligence on the target, then formulate a plan of attack by mimicking the TTP's of the many adversaries we are actively tracking.

By Connor Scott

Senior Consultant

25 Jun 2020

Security, Vulnerabilities and exploits, Security assessment and testing

Introduction

In January, during a simulated attack engagement, Context obtained user level access to an internal network, however the compromised user account had limited access due to a highly locked down environment. A path was identified within the network that would allow the team to obtain privileged access, however it required a method of escalating privileges on client workstations. The workstations all had the latest updates for both Windows 10 and Windows 7 and vendor patches were applied quickly, additionally there was minimal third-party software installed by default, limiting the potential attack surface. While assessing the existing services on the system for avenues of privilege escalation, Zoom's Sharing Service was identified on the shortlist of binaries to triage. After a review of the Zoom service binary, a logic based Local Privilege Escalation vulnerability was identified. This zero-day vulnerability was exploited during the engagement and opened the compromise path leading directly to privileged network accounts.

On February 26, 2020, Context provided Zoom a detailed advisory and POC. The vulnerable version of the Zoom Sharing Service was 4.6.7**.** The issue, which only impacts Windows users, was fully resolved in Zoom release 5.0.4, which was released on May 24, 2020.

The rest of this post contains technical details of the vulnerability and a detailed analysis of the constraints enforced by the service.

Technical Details

When performing a quick triage of a native binary (non .NET) executable for privilege escalation vectors, particularly service binaries, we look for two elements. The first element is an indication that the service executes code or reads/writes data to privileged areas, as services generally run as the SYSTEM account, any sub processes or memory accesses also inherit its privileged context. Looking at the Win32 API imports for function families such as CreateProcess, LoadLibrary, CreateFile or Registry can provide this indication. The second element is an indication that some form of interprocess communications (IPC) exist to control the service. Looking for Shared memory, File Mapping, Socket or Named Pipe functions in the imports can provide this indication. By importing CreateProcessAsUserW and CreateFileMappingW, the Zoom Sharing Service met these criteria and invited a more thorough look.

Note: all images and reverse engineering steps below were performed using GHIDRA on the following file:

Name: CptService.exe

FileVersion: 3.4.2019.0910

MD5: 2a7a76fad78254d39d86504d0dd76dcf

SHA256: 3c8eb269d2a128399146c13a9faf73c5ee3249b04df822a2c37e5e64fa4c6be9

After a session of reversing the CptService.exe binary we had a basic understanding of how the relevant areas worked, in particular:

- The shared file mapping details used for communicating between the user process and the service.
- The expected data and message structure of the IPC.
- The IPC read/action trigger.
- The process execution constraints.

Covering the reverse engineering of all these elements in one article would be quite long, so this article will focus on the part relevant to the vulnerability, i.e. the file execution constraints. Reversing the shared file mapping IPC and the IPC read trigger may be covered in future articles.

In short, the IPC flow can be triggered with the following steps:

- Open the shared file mapping with the pattern "Global\ZOOM_CPTSERVICE_FILE_MAPPING".
- Write a structured IPC message.
- Open the Zoom Sharing Service and send a user defined control code.

This will trigger the service to begin the processing of the message in the shared file mapping IPC buffer.

After some initial tracing from the service entry point through IPC parsing functions we eventually arrive at the function at offset 0x00401317, this function and its child functions handle validation of the supplied IPC parameters. This is where will begin our walkthrough.

Looking at the function graph, we can see a standard branching structure, where child functions are called, and if any of these functions fail, the whole function will return, and the process will not be executed. The interesting functions are called on lines 0x40133d, 0x40134f, 0x40136a and 0x401376. As it is useful to understand the context of these functions to the overall vulnerability, we will summarise their functionality at a high level.

1. 0x40133d - Called function appears to read data from the IPC Memory buffer and returns a pointer if successful, or zero if it fails. We won't cover this in this blog, as the details aren’t really relevant to the issue found
2. 0x40134f - Called function checks that the file exists and that the file permissions allow the requested operation to succeed. In this calling case, it just checks that the file exists. We won't cover this in this blog, as the details aren’t really relevant to the issue found
3. 0x40136a - Called function appears to perform validation of the specified executable file and the DLLs that are imported by it. If we can return success from this function, we can proceed to execute code.
4. 0x401376 - Called function kills existing process in the target Windows Session with executable name matching the supplied filename then launches a new process using the supplied executable filename. At this point there are no more verification checks, so we won't cover this in this blog

In relation to this service, if we can make our supplied file pass the signature and path validation, we can execute code as SYSTEM on the local machine. The vulnerability identified in CVE-2020-9767 allows us to pass this validation and therefore execute arbitrary code of our choosing. As the IPC mechanism is only available locally, this does not pose a threat of direct remote code execution.

If you are interested in the low-level details of this function, continue reading below.

So you care to see how deep the rabbit hole goes?

Having identified that the function 0x404966, called at 0x40136a, performs validation of the supplied executable, we can dig further into it to identify any issues that would allow us to bypass the validation or understand what validation constraints must be met. For ease of reference, from here on we will refer to this function as validate_exe.

Looking at the function from a bird's eye view we can see that there are many places where the execution jumps to a code location that will zero the AL register before executing the epilog, which will effectively flag a failure of verification. There are two jumps that skip this explicit zeroing of AL, allowing a successful result to be returned. These jumps are coloured blue above. The first is at 0x404a07 and explicitly sets AL to 1 before calling the epilog therefore returning valid, the second is at 0x404acb, when the result from the function call at 0x404ab2 is moved into AL and then epilog is called returning true or false based on the output of the function at 0x404651. These two paths give us the opportunity to return successfully from the validate_exe function.

As neither of these blocks are at the start of the function, we will need to understand the constraints imposed on the execution path to reach these blocks. Fortunately, for us both blocks share the same constraints until just before the first block so we get to kill two birds with one stone.

To start with, we can see at line 0x40498a that EDI is set to the address of the filename in the IPC packet and on the following line EBX is set to zero. Then at 0x404997, EDI (the filename pointer) is compared against EBX (zero). If the result matches (EDI == 0), then we return zero from the function indicating it's not a valid executable. So we now know that we need to supply a filename in the IPC packet.

In the next block (0x40499f) we zero EAX, then compare AX (zero) with the word value stored at EDI (the filename pointer), effectively checking that the first character of the filename is not a '\0' (null byte). Note a word comparison is used rather than a byte comparison because the string is a wide (UTF-16) string. If the character is null, then we return zero from the function and it is not a valid executable. So now we know we need to supply at least 1 byte in our filename.

The next 3 blocks (0x4049aa, 0x4049af and 0x4049b9) perform an inline wstrlen and compare the result to 0x208. The first two blocks count the number of bytes in the string, by looping over each wchar, loading it from the location of EAX into CX and comparing it to null. It then increments EAX on each loop by two to account for the wchar. Finally, when CX is null, it breaks the loop and continues to the next block. The next block divides the number of bytes by 2 and compares to 0x208 (decimal 520). If the string is longer than that, the function will return zero indicating it is not a valid executable. So now we know that the filename string must have one or more characters and **520** or fewer characters (not including the null byte).

The next block (0x4049c8) calls PathIsRelativeW with the supplied filename. This will return true (non-zero) if the supplied path is a relative path or zero if the path is an absolute path. If the path is relative, then we return zero from the function and it is not a valid executable. So we now know the path supplied in filepath must be absolute.

We will skip over the next blocks at 0x4049d7, 0x4049df and 0x4049e7, because the values didn't appear to be directly relevant to our path hunting, although the result is passed into the validate_file_signature function.

The next relevant block starts at 0x4049ea (highlighted) and involves a call at 0x4049ee to the function at 0x4032c4, if that call returns zero we return zero from the validate_exe function and our executable is flagged as non-valid. To save time, we won't delve into this function, because there is a fair bit of code that checks the platform for WinTrust support and where it is supported, uses WinTrust to validate the Authenticode signature on the file at the supplied filepath. If it is valid, it checks the issuer of the certificate to ensure it was signed by "Zoom Video Communications, Inc.". If the signature is invalid or the issuer does not match, the function will return failure. On platforms that do not support WinTrust (before 2003, XP) this function returns true in all cases. So now we know that the supplied executable must have a valid signature AND be signed by Zoom on recent operating systems. This significantly limits the pool of options.

Finally, we are at a block that can redirect execution to our first identified opportunity (highlighted green) to return non-zero from the validate_exe function we mentioned all so long ago. We can see that the result of the function call to 0x402e4d at 0x004049fe is checked, if it is non-zero, our first block of interest is executed and we will return 1, and valid, from the validate_exe function. If it is zero, it will continue with further filepath checks. So what does the function at 0x402e4d do? For the sake of this blog, we will save the minutiae. The function appears to get the full path of the current Windows install and checks that it is contained within the supplied filepath string. If it is, it will return 1, if it is not it will return 0.

So to hit our return 1, valid, block, at this point we have the following constraints to meet:

- Supply an absolute filepath to an executable file that will be eventually executed
- The supplied executable must have a valid Authenticode signature, and the Issuer must be "Zoom Video Communications, Inc."
- The supplied file path must be in the Windows install directory, or have the absolute Windows install directory in its filepath

Initially, these seem like challenging constraints, until two things are realised. Firstly, there are subfolders in the Windows install folder where users are able to write, for example C:\Windows\Temp. Secondly, Zoom provides a number of binaries that have valid Authenticode signatures that we can use to pass this signature check. These signed executables load DLLs, so we can effectively use them to 'trampoline' execution to our code.

So this could be one method to achieve code execution, but where is the fun in that? What if user write permissions in the Windows install directory are locked down? What if we want to execute from arbitrary folders, say for example C:\Users\Public\Context for vanity purposes? To do this, we need to press on, which for the sake of brevity will be covered in a future blog.

Vendor Response

The identified vulnerability was disclosed to Zoom in late February 2020. Context and Zoom staff followed coordinated disclosure processes to confirm the issue was resolved and to ensure their respective clients using this software were protected.

Zoom's advisory for this issue may be found at: https://support.zoom.us/hc/en-us/articles/360044350792-Security-CVE-2020-9767.

Context's advisory for this issue may be found at: https://www.contextis.com/en/resources/advisories/cve-2020-9767

Mitigations

There are several recommendations to help mitigate this vulnerability and we welcome questions and queries.

Update Zoom

Firstly, as with all software we strongly recommend that any application be updated when possible. Given the current climate, updating via corporate deployment services may be more difficult. As such, teams should be encouraged to update Zoom as well as any other applications directly on the Internet if applicable to their network architecture. Zoom 5.0.4 was verified to remediate this vulnerability and working with the Zoom security team, the underlying flaws have been addressed. Later versions should also include these fixes.

Should IT administrators be unable to upgrade Zoom to version 5.0.4 and above, there are two other options for mitigating this particular vulnerability.

Implement Application and DLL Whitelisting

Application and DLL whitelisting has proven to be an effective defence in depth measure to combat the execution of untrusted executables and DLLs. When implementing an effective application and DLL whitelisting program, ensure that folder allow rules such as C:\Windows\* are not included as users may write files to and execute files in subfolders of this folder. In addition, a baseline should be performed capturing Zoom's installed binaries and DLLs, these should be whitelisted, effectively restricting other unapproved software from piggybacking on execution flow.

Disable Zoom Sharing Service

Through testing in a laboratory environment and in conjunction with the client during the simulated attack, stopping the 'CptService/Zoom Sharing Service' did not appear to have a negative impact during general purpose use. Screen sharing, desktop controls and all other functionality appeared to function, as such it was the recommendation as an immediate remediation to disable this service. Upon further investigation and discussion with the Zoom Security team, this service was identified as being used to potentially share elevated windows dialogs.

Generally, users sharing elevated windows would be initially privileged users and may inherit the integrity level when launching zoom (dependent on local security controls). As such, this service could be disabled with minimal impact to most of the user base and attack surface immediately remediating this issue.

About Connor Scott

Senior Consultant

Connor is part of our Assurance team and is based in our Melbourne, Australia office, he specialises in Red Teaming, Reverse Engineering, Software and Embedded Device Vulnerability Research, Infrastructure Assessments and Code Reviews.