Fuzzing Microsoft's RDP Client using Virtual Channels

Valentino Ricotta
ricotta.valentino@gmail.com

Thalium

Abstract. The *Remote Desktop Protocol* (RDP) is a proprietary protocol designed by Microsoft that allows a user to connect to a remote computer over the network with a graphical interface. Though server-side security has often been studied, the security of RDP client applications remains more peripheral. For all that, the richness of the protocol and the width of the attack surface make RDP clients valuable fuzzing targets.

This article describes how to leverage the WTS API to setup a fuzzing architecture for Microsoft's RDP client based on WinAFL, and suggests a methodology targeting the *Virtual Channels* abstraction layer. Throughout a few channels such as those dedicated to sound redirection, clipboard, printers or smart cards, several bugs were identified, including two CVEs: an Information Disclosure and a Remote Code Execution.

1 Introduction

The *Remote Desktop Protocol* (RDP) is a proprietary protocol designed by Microsoft that allows a user to connect to a remote computer over the network with a graphical interface. Its use around the world is very widespread; some people, for instance, use it often for remote work and administration.

Although RDP dates back to Windows NT 4.0 (then formerly known as Terminal Services) and many vulnerabilities have been found in it over the years, it is in 2014 that researchers from Tripwire, Inc. publish one of the first works on RDP fuzzing [2]. In 2019, Eyal Itkin of Check Point Research published work targeting RDP clients in general [5], that led to 16 major vulnerabilities in open-source clients and a path traversal attack in Microsoft's client that also impacted the Hyper-V manager.

During a conference talk at *Blackhat Europe 2019* [4], Chun Sung Park, Yeongjin Jang, Seungjoo Kim and Ki Taek Lee explained that they managed to exploit an RCE inside Microsoft Windows' RDP client by fuzzing the *Virtual Channels* of RDP using WinAFL [6]. We thought they achieved encouraging results that deserved to be prolonged. The objective

was to go further, by coming up with a general methodology for attacking these *Virtual Channels* that would widen the fuzzing surface.

This work was conducted as part of my second-year engineering internship at Thalium, where I spent time studying and reverse engineering Microsoft RDP, learning about fuzzing, and looking for vulnerabilities.

In parallel, in 2021, researchers from CyberArk have published some of the work they conducted on fuzzing RDP [25] as well. Though they also used WinAFL and faced similar challenges, their fuzzing approach somewhat differs from the one presented here. [1]

This article first presents a few elements that are necessary to understand how the *Remote Desktop Protocol* works. Then, it describes the architecture that was set up and the methodology used for fuzzing Microsoft's RDP client with WinAFL. Finally, some results will be presented in more detail, especially the vulnerabilities that were found.

1.1 Why search for vulnerabilities in the RDP *client*?

An example of an RDP client attack scenario is given in the *Blackhat Europe 2019* conference talk [4]. The authors' research was driven by the idea that North Korean hackers would allegedly carry out attacks through compromised RDP servers acting as proxies. By setting up a *honeypot* to which they would connect, one could "hack them back", assuming a vulnerability in the client is known.

Vulnerabilities in the RDP client can also lead to guest-to-host virtual machine escape attacks in Hyper-V. Indeed, since the Hyper-V manager internally uses RDP to implement features such as screen sharing, remote keyboard or synchronized clipboard, it inherits its potential security flaws.

Aside from these motives, most of vulnerability research seems to be focused on server implementations; CVEs in the RDP client are more scarce, even though the attack surface is as large as the server's.

2 Study of the *Remote Desktop Protocol*

This article only presents a few elements of RDP that are needed to understand how to fuzz *Virtual Channels*. Other resources such as blog articles or the Microsoft specification itself [14, 24] explain the protocol in more detail.

1. Some major differences are that they implemented multi-input fuzzing, and that they also fuzzed the RDP server.

Microsoft has its own implementation of RDP (client and server) built in Windows. There also exist alternate implementations of RDP, such as the open-source FreeRDP [1]. By default, the RDP server listens on TCP port 3389. UDP is also supported to improve performance for certain tasks such as bitmap or audio delivery. In Windows 10, the main file of interest for most of the client logic is `mstscax.dll`.

Basic, core functionalities of an RDP client include receiving desktop bitmaps from the server, and sending keyboard and mouse inputs to the server. However, a lot of other information can be exchanged one way or the other: sound, clipboard, support for special types of hardware, etc. This information goes through what Microsoft call *Virtual Channels*.

2.1 *Virtual Channels*

Virtual Channels (or just *channels*) are an abstraction layer in RDP used to generically transport data. They can add functional enhancements to an RDP session. The *Remote Desktop Protocol* provides multiplexed management of multiple *virtual channels*. Each individual channel behaves according to its own separate logic, specification and protocol. Microsoft specifies dozens of official channels [7].

The RDP stack consists of several layers, sometimes with multiple levels of encryption. *Virtual Channels* operate on the MCS (*Multipoint Communication Service*) layer (fig. 1). Thanksfully, Windows provides an API called the WTS API [20] to interact with this layer, which allows to open, read from and write to a channel. This comes convenient for writing a fuzzing harness.

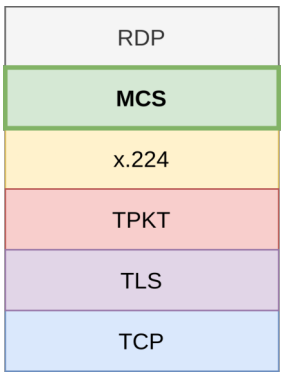


Fig. 1. Remote Desktop Procol stack.

There are two types of *Virtual Channels*: static ones and dynamic ones. *Static Virtual Channels* (SVC) are negotiated during the connection phase. They are opened once for the session and are identified by a name up to 8 bytes. By default, the RDP client asks to open the four following SVCs:

— `RDPSND`: audio redirection from the server to the client;
— `CLIPRDR`: two-way clipboard redirection/synchronization;
— `RDPDR`: filesystem redirection (and more...);
— `DRDYNVC`: support for dynamic channels.

Dynamic Virtual Channels (DVC) are built on top of the `DRDYNVC` SVC, which manages them. They can be opened and closed on the fly during an RDP session. They are especially used by developers to create extensions. Microsoft provides a fair amount of official DVCs (touch and pen input, geometric rendering, display configuration, telemetry, microphones, webcams, PnP redirection...), some of which are automatically enabled.

In conclusion, both types of channels are great targets for fuzzing. Each channel behaves independently, has a different protocol parser, different logic, lots of different structures, and can hide many bugs. What is more, channels that are open by default are an even more interesting target risk-wise, because any vulnerability found in these will directly impact most clients.

3 Architecture for fuzzing the RDP client

Since there was little to no information publicly available about the fuzzer presented at *Blackhat Europe 2019* [4], the choice was made to implement a new architecture for fuzzing the RDP client. We decided however to take inspiration from two elements: the use of WinAFL, and the network-level approach for the harness.

3.1 WinAFL: presentation and choices

WinAFL [6] is a Windows fork of the popular mutational fuzzing tool AFL [12]. It works by continuously sending and mutating inputs to a target program in order to make it behave unexpectedly (and hopefully crash). Mutations are repeatedly performed on samples which must initially come from a *corpus* (a set of input files or *seeds*).

As described in *The Art, Science, and Engineering of Fuzzing* by Manès et al. [10], AFL and its descendants are *grey-box fuzzers*, which means they are *feedback-based* or *coverage-guided*. Coverage-guided fuzzers

instrument the target binary to compute, for each execution, the branch coverage (for example, which basic blocks were visited). This technique allows the fuzzer to explore new paths within the target binary, and with which inputs they are reached.

In order to achieve coverage-guided fuzzing, WinAFL provides several instrumentation modes: dynamic instrumentation using DynamoRIO, Intel PT and Syzygy. Because Intel PT has limitations within virtualized environments and Syzygy is restricted to 32-bit binaries with full PDB symbols, the adopted mode was DynamoRIO.

DynamoRIO [29] is a dynamic binary instrumentation framework. It provides an API to deal with black-box targets, which WinAFL can use to instrument the target binary (in particular, monitor code coverage at run time).

Finally, when fuzzing, killing and restarting the RDP client each iteration is unwanted as it would add enormous overhead. To alleviate that, DynamoRIO provides several *persistence modes* that dictate how the fuzzer should exactly loop on the target function:

— *Native persistence*: measure coverage of the target function, and on *return*, reload context and redirect execution back to the start of the target function;
— *In-app persistence*: let the program loop naturally, and coverage will reset each time in the `pre_loop_start_handler`, inserted right before the target function;
— "No-loop" mode: similar to in-app persistence, with the advantage of stopping coverage on *return*. This mode was found by reading WinAFL's codebase and does not seem documented.

The "no-loop" mode was chosen as it seems adapted to the context of network fuzzing, while still producing code coverage limited to the part of interest inside the RDP client (the one that handles the channel being fuzzed). Figure 2 summarizes, in a simplified manner, the fuzzing process using WinAFL's "no-loop" mode.

One should also mind the importance of the `-thread-coverage` option in DynamoRIO, which limits code coverage measurement to the thread that triggered the target function. Forgetting this option will negatively impact the fuzzer's *stability* metric, because coverage will include heavy noise from other threads's activity in the RDP client, rendering the fuzzing very random.

GFlags was also enabled with PageHeap [13]. Applying the `/full` option on `mstscax.dll` asks Windows to place an unreachable page at the

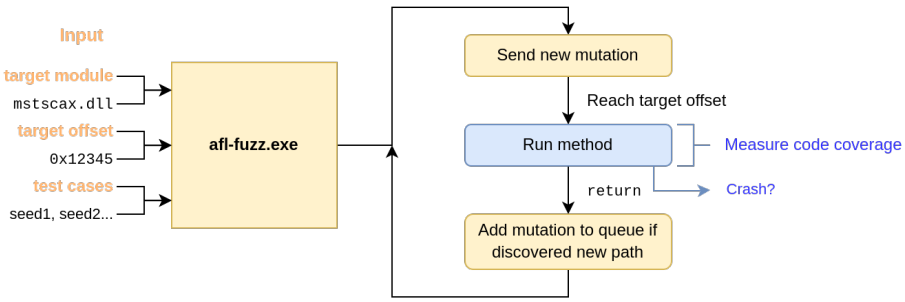


Fig. 2. Fuzzing process with WinAFL in "no-loop" mode.

end of each heap allocation. Therefore, as soon as there is an out-of-bounds access, the client will crash.

3.2 Setting up WinAFL for network fuzzing

By default, WinAFL writes mutations to a file that should be passed as an argument to the target binary. The target being a network client, we can either fuzz it through the network by sending packets, or try to harness directly the functions inside the client that handle incoming packets and modify the packets in memory, for instance with a snapshot-based approach.

The choice was made to make the harness act like a server that sends mutations to the client over the network. This requires developing a server-side harness, and then adapting WinAFL so that it redirects the mutations over to the harness. Although it may seem like it would slow down the fuzzer, sending mutations over the network is actually not a bottleneck at all in terms of fuzzing speed (at least locally).

The harness runs in parallel of the RDP server. It listens on a given TCP port and waits to receive an input mutation. It optionally processes it, and sends the mutation back to the RDP client through a specified *Virtual Channel*. This is easily implemented using the aforementioned WTS API. Finally, WinAFL is modified to send mutations to the harness via TCP by changing the `write_to_testcase` function. The fuzzer architecture is drawn figure 3.

In the *Blackhat Europe 2019* conference talk [4], the authors used two virtual machines (one for the client and one for the server). Indeed, it is not normally possible, by design, to connect to a local RDP server on the same machine. The RDPWrap [26] tool allows to bypass this limitation: the fuzzer therefore fits in a single virtual machine.

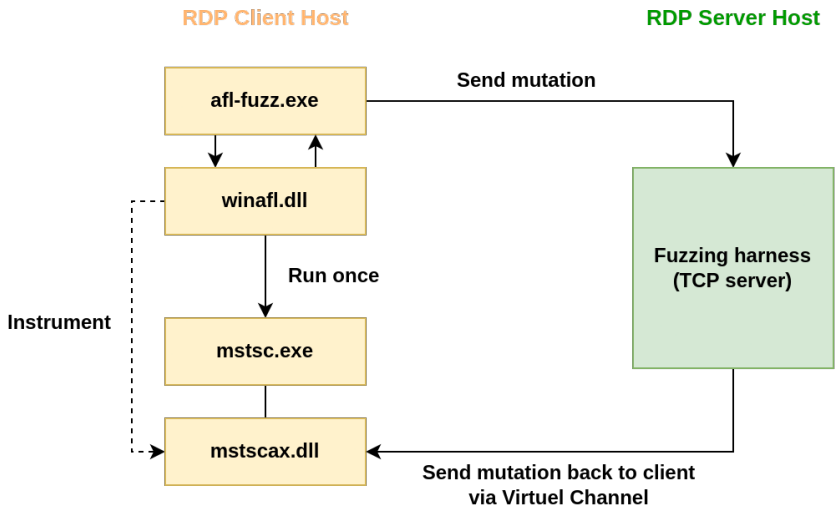


Fig. 3. Architecture of the fuzzer.

4 Fuzzing methodology

The harness is functional, but before actually fuzzing, one needs to agree on what to attack and which approach to use. This section will illustrate the different steps with the example of the `RDPSND` channel (sound redirection).

4.1 Attacking a channel

Once the target channel is selected, two elements are concretely needed to start fuzzing: a *target offset* (address of the target function), and an *initial corpus*.

The first step is to read the channel specification provided by Microsoft (for `RDPSND`: [15]). It describes the channel's functioning quite exhaustively, with all the different message types, structures, protocol diagrams, as well as many examples of PDU (*Protocol Data Unit*) hexdumps. These are great mutation seeds for the fuzzer.

Then, the RDP client can be reverse engineered to locate where incoming PDUs in the channel are received and processed. PDB symbols, strings and magic numbers are often enough to identify these channel handlers. For instance, in `RDPSND`, the target method is rather straightforward (fig. 4). In case it is not enough, one can capture code coverage at the moment a PDU is sent to the target channel. This can be achieved

with Frida and `frida-drcov` [23,30]. The Lighthouse IDA plugin [11] then allows to visualize the code coverage.

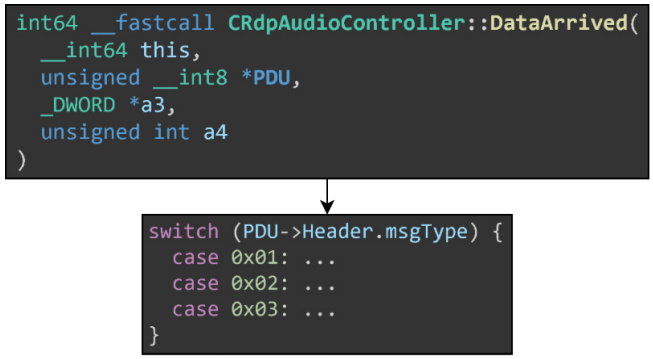


Fig. 4. Target method for the `RDPSND` channel.

4.2 Different fuzzing strategies

Once the target offset is known, should we start fuzzing naively with the seeds gathered from the specification?

There is still one main problem that arises: the problem of *stateful fuzzing* in a network context. Indeed, the RDP client can be modelled by a complex state machine. This state machine could be subdivided in several smaller state machines for each channel, but which would remain fundamentally complex to characterize.

We suggest two main stategies: a "naive" one, and one that partially addresses the mentioned problem, but at some cost. However, neither strategy entirely addresses the stateful fuzzing issue: it was estimated it would require significant additional work, and the initial plan was to be able to fuzz many channels with a lesser effort.

Mixed message type fuzzing. This is the "naive" strategy: the *seeds* gathered from the specification are used "as is", and without modifying the harness any further: it sends back the raw mutations to the client. Figure 5 describes the structure of an `RDPSND` PDU header.

Since the mutation seeds include the header, the fuzzer will also mutate it, including the `msgType` field. Therefore, the RDP client will receive a lot of different message types, in a rather random order. This is an interesting

Fig. 5. `SNDPROLOG` header of an `RDPSND` PDU.

approach because sending a sequence of PDUs of different types in a certain order could help the client enter a state in which a bug is triggered.

A drawback of this strategy is that crash analysis becomes more difficult. Since a larger space of PDUs is covered, and thus a larger space of states, there is a higher chance that a bug originates from a complex sequence of states. When WinAFL detects a crash and saves the associated mutation, there is no guarantee whatsoever that reproducing the bug is feasible with this mutation only. If the bug does not reproduce, it is probably because it is rather a sequence of PDUs that crashed the client, and not just a single PDU. However, understanding which sequence of PDUs made the client crash is often difficult and requires a more in-depth analysis.

Fixed message type fuzzing. This time, WinAFL will operate only on the message's body. In particular, the `msgType` will not be mutated, implying a fuzzing campaign should be started for each individual message type (there are 13 in `RDPSND`). For instance, one can target specifically the *"Server Audio Formats and Version"* PDUs in `RDPSND` (figure 6).

This strategy is still vulnerable to the presence of "stateful bugs", but generally less than in mixed message type fuzzing, because the state space is smaller. However, it requires some preparation: cutting the seeds' headers, and writing a specific wrapper for each channel at harness-level to reconstruct the header. In certain cases, it may also be useful to identify the methods in the binary that handle each message type (for instance in the `CLIPRDR` channel, where these methods are called asynchronously).

We conclude both fuzzing approaches should be taken into consideration. The first one can find more uncommon bugs, but which are sometimes very hard to analyze. The second one needs a bit more effort to set up, but allows to go more in depth in each message type's logic, and the bugs found are usually easier to reproduce.

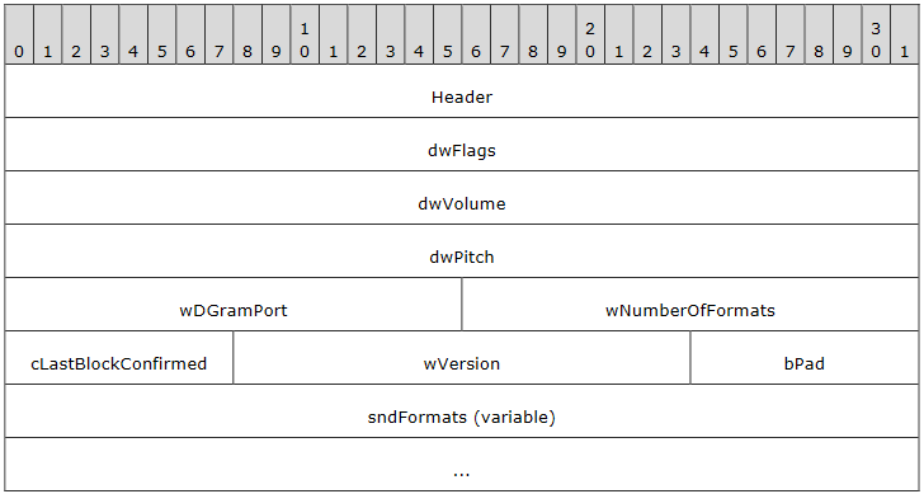


Fig. 6. `RDPSND` *Server Audio Formats and Version* PDU structure.

4.3 Analyzing crashes

As mentioned, analyzing a crash can range from easy to nearly impossible. When WinAFL finds a crash, the only thing it does is save the associated mutation to a file. From there, there are two possibilities:

— the crash is successfully reproduced. In this case, what is only left is to reverse to understand the root cause, analyze risk, and maybe grow the crash into a bigger vulnerability;
— the crash cannot be reproduced. In this case, one can try working their way through "blindly" by dissecting the guilty payload...

For analysis purposes, a modification of WinAFL to log more information about crashes (exception address, module, timestamp and exception information) proved to be of great use. This way, even when the crash cannot be reproduced, one can still locate where the crash occurred. They usually occur in `mstscax.dll`, but some bugs may happen in other modules. It is worth noting a crash in an "unknown module" could mean the execution flow was redirected.

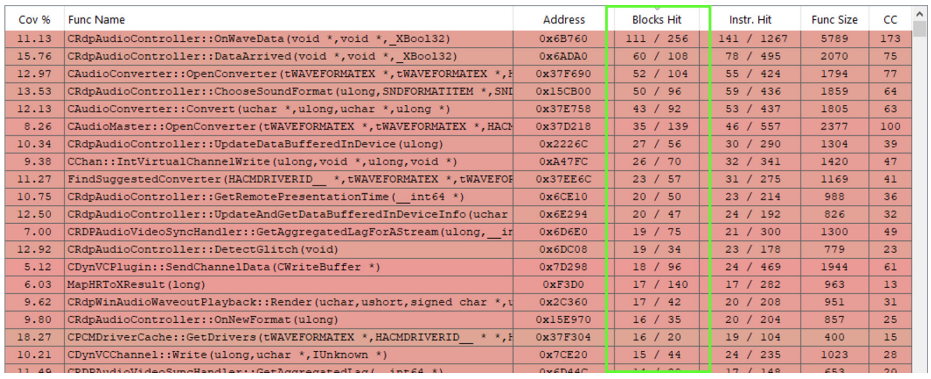
4.4 Assessing fuzzing quality

Knowing when to stop fuzzing a channel exactly is not an easy task. As during any fuzzing campaign, the number of paths found over time always tends to reach a certain plateau. But although a plateau is reached, waiting a few additional hours could very well lead to a lucky strike in

which a new mutation is found, in turn snowballing into dozens of new paths.

The main criterion to take into account is code coverage quality. To help with assessing code coverage quality, a modification in WinAFL adds an option that saves all the encountered basic blocks at each fuzzing iteration, and logs them into a file if the iteration produced a new path. For each new path, the corresponding basic block trace can be converted in the `Mod+Offset` format in order to be visualized with Lighthouse [11].

Figure 7 shows Lighthouse's visualization of the obtained code coverage for the `RDPSND` channel. The proportion of blocks hit in each "audio" function is a good indicator of quality, although it seldom reaches 50% because there is a large proportion of error-handling blocks that are never visited.



Cov %	Func Name	Address	Blocks Hit	Instr. Hit	Func Size	CC
11.13	CRdpAudioController::OnWaveData(void *,void *,_XBool32)	0x6B760	111 / 256	141 / 1267	5789	173
15.76	CRdpAudioController::DataArrived(void *,void *,_XBool32)	0x6ADA0	60 / 108	78 / 495	2070	75
12.97	CAudioConverter::OpenConverter(tWAVEFORMATEX *,tWAVEFORMATEX *,H	0x37F690	52 / 104	55 / 424	1794	77
13.53	CRdpAudioController::ChooseSoundFormat(ulong,SNDFORMATITEM *,SNI	0x15CB00	50 / 96	59 / 436	1859	64
12.13	CAudioConverter::Convert(uchar *,ulong,uchar *,ulong *)	0x37E758	43 / 92	53 / 437	1805	63
8.26	CAudioMaster::OpenConverter(tWAVEFORMATEX *,tWAVEFORMATEX *,HACM	0x37D218	35 / 139	46 / 557	2377	100
10.34	CRdpAudioController::UpdateDataBufferedInDevice(ulong)	0x2226C	27 / 56	30 / 290	1304	39
9.38	CChan::IntVirtualChannelWrite(ulong,void *,ulong,void *)	0xA47FC	26 / 70	32 / 341	1420	47
11.27	FindSuggestedConverter(HACMDRIVERID__ *,tWAVEFORMATEX *,tWAVEFOR	0x37EE6C	23 / 57	31 / 275	1169	41
10.75	CRdpAudioController::GetRemotePresentationTime(__int64 *)	0x6CE10	20 / 50	23 / 214	988	36
12.50	CRdpAudioController::UpdateAndGetDataBufferedInDeviceInfo(uchar	0x6E294	20 / 47	24 / 192	826	32
7.00	CRDPAudioVideoSyncHandler::GetAggregatedLagForAStream(ulong,__ir	0x6D6E0	19 / 75	21 / 300	1300	49
12.92	CRdpAudioController::DetectGlitch(void)	0x6DC08	19 / 34	23 / 178	779	23
5.12	CDynVCPlugin::SendChannelData(CWriteBuffer *)	0x7D298	18 / 96	24 / 469	1944	61
6.03	MapHRToXResult(long)	0xF3D0	17 / 140	17 / 282	963	13
9.62	CRdpWinAudioWaveoutPlayback::Render(uchar,ushort,signed char *,u	0x2C360	17 / 42	20 / 208	951	31
9.80	CRdpAudioController::OnNewFormat(ulong)	0x15E970	16 / 35	20 / 204	857	25
18.27	CPCMDriverCache::GetDrivers(tWAVEFORMATEX *,HACMDRIVERID__ * *,H	0x37F304	16 / 20	19 / 104	400	15
10.21	CDynVCChannel::Write(ulong,uchar *,IUnknown *)	0x7CE20	15 / 44	24 / 235	1023	28

Fig. 7. Code coverage for the `RDPSND` channel fuzzing campaign in Lighthouse.

Skimming through the functions, one can assess whether they are satisfied with the fuzzing campaign. However, this requires having reverse engineered the channel enough to have a good depiction of its architecture and inner workings in mind; more specifically, to know what are all the functions and basic blocks of interest.

5 Results

This section presents some results in a few channels where fuzzing campaigns were attempted.

Table 1 synthesizes the *fuzzing level* of each channel and the number of bugs found. *Fuzzing level* is a subjective scale to assess how much and

Channel	Description	*Fuzzing level*	Bugs
RDPSND	Audio redirection	2	1
CLIPRDR	Clipboard	1	1
DRDYNVC	Dynamic channels manager	0	–
RDPDR	Filesystem redirection, printers, smart cards...	2	3

Table 1. Results of the fuzzing campaigns.

how well each channel was fuzzed: 0 if fuzzing failed, 1 if fuzzing could have been done better or more in depth, and 2 if coverage was satisfying enough (of course, this does not imply at all that the channel is exempt from any bugs). In particular, three channels were effectively fuzzed, and one (DRDYNVC) could not be fuzzed.

5.1 RDPSND

RDPSND is a static virtual channel that transports audio data from server to client, so that the client is able to play sound originating from the server. It is open by default. Most of the message types referenced in the specification [15] were fuzzed. Each message type was fuzzed for hours and the channel as a whole for days. Code coverage is decent.

One crash was found that is not further exploitable, but that will still be detailed as it is a good example of "stateful bug".

Out-of-Bounds Read in RDPSND.[2] The crash happened upon receipt of a *Wave2* PDU, inside CRdpAudioController::OnWaveData. Dissecting the PDU (listing 1) does not reveal anything particularly shocking right away. On a purely semantic level, fields that could be good candidates for a crash are wFormatNo or cBlockNo, because they may index an array.

```
0d 00 10 00   Header
16 a1         wTimeStamp
0f 00         wFormatNo
20            cBlockNo
f5 00 00      bPad
c2 b8 b3 0d   dwAudioTimeStamp
de 20 be ef   Data
```

Listing 1. Out-of-Bounds Read bug in RDPSND: guilty PDU dissection.

Reversing the OnWaveData function (listing 2) allows to understand the bug. The attacker controls wFormatNo (unsigned short), and the crash

2. CyberArk also found this bug and described it in their own article [25].

occurs when computing `targetFormat`. The out-of-bounds crash is quite obvious; however, manually replaying the malicious PDU has no effect. This is a case of "stateful bug" in which a sequence of PDUs crashed the client, and only the last PDU is known.

```
wFormatNo = PDU->Body.wFormatNo;

// Has wFormatNo changed since the last Wave PDU?
if (wFormatNo != this->lastFormatNo) {
  // Load the new format
  if (!CRdpAudioController::OnNewFormat(this, wFormatNo)) {
    // Error, exit
  }
  this->lastFormatNo = wFormatNo;
}

// Fetch the audio format of index wFormatNo
savedAudioFormats = this->savedAudioFormats;
targetFormat = *(AudioFormat **)(savedAudioFormats + 8 * wFormatNo);

wFormatTag = targetFormat->wFormatTag;
```

Listing 2. Vulnerable piece of the `OnWaveData` function in `RDPSND`.

No length checking is performed here on `wFormatNo`, but there is actually a check inside the `OnNewFormat` function. In order to trigger the bug, the condition has to be skipped over, and for that, `wFormatNo` should be equal to the last one that was sent (`this->lastFormatNo`). The answer to the problem lies in the *Server Audio Formats and Version* PDU (figure 6). This PDU is used by the server to send a list of supported audio formats to the client. The client will save this list of formats in `this->savedAudioFormats`. Therefore, the bug can be triggered by sending a *Format* PDU between two *Wave* PDUs to make this list smaller. More specifically:

1. Send $n > 1$ formats to the client through a *Format* PDU.
2. Send a *Wave* PDU with `wFormatNo` set to n.
3. Send a new *Format* PDU with $k < n$ formats: the format list is freed and reconstructed.
4. Send the same *Wave* PDU than in step 2: since `lastFormatNo` is n, the length check inside `OnNewFormat` is bypassed and the out-of-bounds read triggered.

Although this bug cannot be grown into an actual vulnerability because the memory read cannot be leaked back to the server, it highlights how "mixed message type fuzzing" can help find new bugs. WinAFL managed to find a sequence of PDUs which bypasses a certain condition to trigger a crash that could have been otherwise overlooked.

5.2 CLIPRDR

`CLIPRDR` is a static virtual channel dedicated to the synchronization of the clipboard between the server and the client. It allows to copy and paste several types of data (text, images, files...) from server to client and vice versa. It is open by default.

Unlike most other channels, `CLIPRDR` is modelled by an actual state machine (documented in [16]) and includes proper state verification. Indeed, each PDU sub-handler (logic for a certain message type) calls the `CheckClipboardStateTable` function prior to anything. This function tracks and ensures the client is in the correct state to process the PDU. If it is not, it just drops the message and does not do anything. This is a concern for two major reasons:

1. In mixed message type fuzzing, very few PDU sequences would make sense and pass the state checks, therefore a lot of the fuzzing effort would go to waste.
2. Fixed message type fuzzing would not work either, or it would require finding a way to bring the client to the right state before each iteration and for each message type, which is not easy to characterize and implement.

`CLIPRDR` comes with another surprise: incoming PDUs are dispatched asynchronously inside the `CClipRdrPduDispatcher::DispatchPdu` function. The PDU sub-handling logic is thus run in a different thread, which renders DynamoRIO's `thread_coverage` option useless.

The choice was made to perform blind mixed message type fuzzing (without thread coverage). This weaker strategy still allowed to identify a bug.

Arbitrary Malloc Denial-of-Service in `CLIPRDR`. This bug showcases a golden rule of fuzzing: that it is not only about crashes and that side effects of fuzzing on a system can also reveal bugs. While blindly fuzzing the channel, the virtual machine would always end up freezing entirely, which required hard rebooting it. This was due to memory overcommitment in the RDP client: it would, at some point, very quickly fill up the system's RAM until reaching "death by swap".

Narrowing down the candidates for a malicious PDU was made possible by purposefully slowing down the harness. The PDU listing 3 is a minimal reproduction case of the bug: a *Lock Clipboard Data* PDU which only contains a `clipDataId` field.

```
0a 00         msgType
00 00         msgFlags
04 00 00 00   dataLen
01 69 63 6b   clipDataId
```

Listing 3. DoS in `CLIPRDR`: guilty PDU dissection.

In the `CClipBase::OnLockClipData` function, this field is used in a `SmartArray` object, and the attacker-controlled value (an unsigned 32-bit integer) is eventually used in a memory allocation (listing 4). This leads to a `malloc` of size $8 \times (32 + \text{clipDataId})$, which means at maximum a little more than 32 GB.

```
v5 = operator new(saturated_mul(32 + clipDataId, 8));
```

Listing 4. Vulnerable piece in `CLIPRDR`: arbitrary memory allocation in `DynArray::Grow`.

Risk-wise, on systems with a moderate amount of RAM, this is a case of remote system-wide denial of service; less impressive on a client than on a server, but may still be dangerous. Moreover, the malicious payloads can be sent in small increments to adapt to the amount of RAM on the victim's system (allocating too much at once will trigger `ERROR_NOT_ENOUGH_MEMORY`).

5.3 DRDYNVC

`DRDYNVC` is a static virtual channel dedicated to the support of dynamic virtual channels (DVC). It allows to create, open and close DVCs, and data transported through DVCs is actually transported over `DRDYNVC`, which acts a wrapping layer. It is open by default.

Unfortunately, we ran into some complications when trying to harness the channel. When opening a channel through `WTSVirtualChannelOpen`, WTSAPI32 eventually ends up performing a Remote Procedure Call. The endpoint of the RPC is located in `termsrv.dll`'s function `RpcCreateVirtualChannel`, which leads to the `CUMRDPConnection::CreateVirtualChannel` function inside `rdpcorets.dll` (listing 5).

```
if ( !_stricmp("DRDYNVC", channel_name)
  || !_stricmp("rdpgrfx", channel_name)
  || !_stricmp("rdpinpt", channel_name)
  || !_stricmp("rdpcmd", channel_name)
  || !_stricmp("rdplic", channel_name)
  || !_stricmp("Microsoft::Windows::RDS::Graphics", channel_name) )
{
```

```
8   error_code = 0x80070005;
9   goto LABEL_58;
10 }
```

Listing 5. Channel opening blacklist in `CreateVirtualChannel` (`rdpcorets.dll`).

`DRDYNVC` is therefore blacklisted from being opened via the WTS API, along with some other channels. Other approaches such as patching the DLLs did not yield favorable results, hence why this channel was not further explored.

5.4 RDPDR

Last but not least, `RDPDR` is the static virtuel channel dedicated to redirecting access from the server to the client's file system. It is open by default. It is also the base channel that hosts several sub-extensions such as the *smart card extension*, the *printing extension* or the *ports extension*. Figure 8 shows the architecture of the channel in `mstscax.dll`.

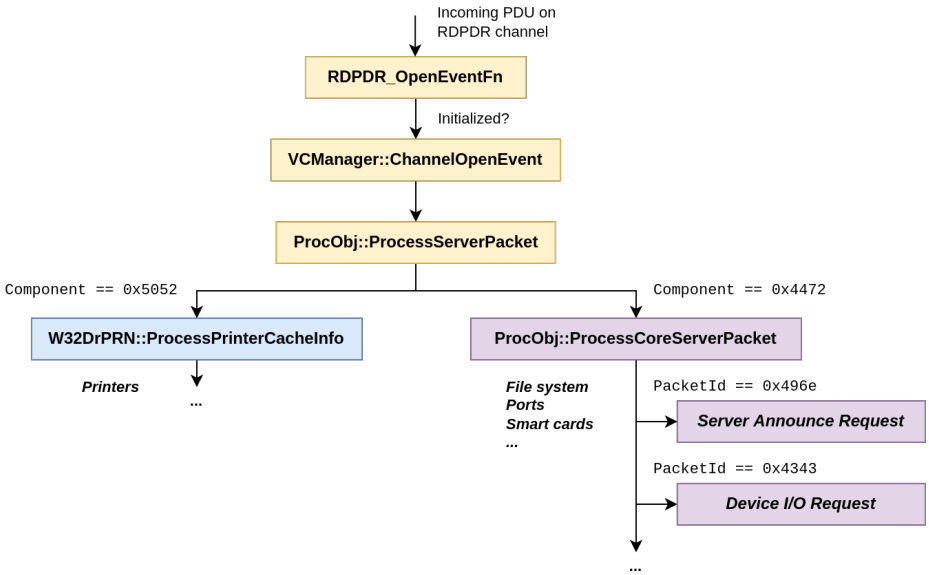


Fig. 8. `RDPDR` channel architecture in `mstscax.dll` and header structure.

In this channel, we encountered a difficulty: the client closes the channel as soon as anything goes wrong while handling an incoming PDU (length checking failure, unrecognized enum value...). The choice was made to

patch the DLL to get rid of this measure. However, one should be very careful while patching a fuzzing target. Since the patch modifies the client's behavior, real bugs in the RDP client will only constitute a subset of the bugs found in the patched DLL.

The channel was fuzzed as a whole, including the sub-protocols (printer, smart cards...). Three bugs were identified.

Arbitrary Malloc Denial-of-Service in `RDPDR`. This first bug is highly similar to the one found in `CLIPRDR`, so it will not be detailed further. A malicious *Device I/O Request* PDU of sub-type *Device Control Request* can trigger an arbitrary memory allocation up to 4 GB inside `W32SCard::MsgIrpDeviceControl`.

Remote Heap Leak in `RDPDR`. This vulnerability resides in `RDPDR`'s printer sub-protocol. It was reported to Microsoft, which assigned it CVE-2021-38665 [22,27], and assessed it as *Information Disclosure* of *Important* severity.

Similarly to some previous bugs, the crashes WinAFL found were not what led to discover this bug. Rather, it was the prolonged fuzzing and the millions of executions that unveiled unexpected side effects the server could have on the client's system. After a while, every time it was run, the RDP client would start consuming a lot of RAM, until eventually hanging the whole system.

The reason was that upon starting, the client would keep iterating on registry keys inside `HKCU\Software\Microsoft\Terminal Server Client\Default\AddIns\RDPDR`, and the more keys, the worse the memory consumption. This fact alone is already very annoying for a client: it is more serious than a simple crash or arbitrary memory allocation. Since the bug is persistent, it entirely prevents one from using their RDP client ever again, unless they specifically know how to fix the problem by deleting the correct keys in the registry.

Figure 9 shows the guilty keys inside the registry. Their names are actually WinAFL mutations interpreted as UTF-16. However, what catches the eye is that these key names are of quite variable length, and may suggest an out-of-bounds of some sort.

The *Add Printer Cachedata* PDU type, found in the printer subprotocol specification [17], is responsible for creating these registry keys. Reversing the `W32DrPRN::AddPrinterCacheInfo` function (listing 6) shows that the key name (`PrinterName`) is entirely controlled.

```
__int64 W32DrPRN::AddPrinterCacheInfo(DR_PRN_ADD_CACHEDATA * PduBody
    , unsigned int PduBodyLength) {
  // ...
  const WCHAR * PrinterName = (const WCHAR *)((char *)PduBody + 24 +
      PduBody->PnPNameLen + PduBody->DriverNameLen);
  RegCreateKeyExW(
    HKEY_CURRENT_USER,
    L"Software\\Microsoft\\Terminal Server Client\\Default\\AddIns\\
        RDPDR",
    0, 0, 0, 0xF003F, 0, &hKey, &dwDisposition
  );
  RegCreateKeyExW(hKey, PrinterName, 0, 0, 0, 0xF003F, 0, &phkResult
      , &dwDisposition);
  RegSetValueExW(phkResult, L"PrinterCacheData", 0, 3, (const BYTE
      *)PduBody, TotalLen);
  // ...
}
```

Listing 6. `W32DrPRN::AddPrinterCacheInfo` function in the `RDPDR` channel.

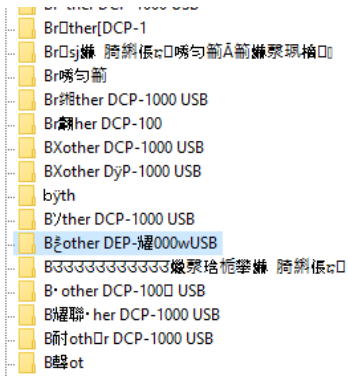


Fig. 9. Mutated key names inside the registry while fuzzing `RDPDR`.

The bug comes from the fact that the server can send a non-null terminated wide string as `PrinterName`. It is naively passed to `RegCreateKeyExW`, which will create the key assuming its name stops at the first wide null-byte (`0x0000`).

For example, by sending a PDU with the `PrinterName` field set to `Brother DCP-1000 USB` and without terminating wide null byte, the client may create a key with the following name (in hexadecimal): `42 00 72 00 6F 00 74 00 68 00 65 00 72 00 20 00 44 00 43 00 50 00 2D 00 31 00 30 00 30 00 30 00 20 00 55 00 53 00 42 00 20 00 61 62 63 64 8D 28 0F 90 00 40 00 90 D8 92 60 C1 FB 7F`.

One can obtain these values by reading the key name from the registry and encoding it as UTF-16. This reveals in particular that some heap bytes were leaked at the end of the name, including an address: `0x7FFBC16092D8` (a *vtable* pointer in `mstscax.dll`). This address leak was quite consistent across different environments, allowing to weaken ASLR and calculate `mstscax.dll`'s base address. In case there is no valuable leak, the attacker can try sending the PDU again, and do so as many times as they want.

In order to repatriate the leaks back to the server, there exists in the protocol a *Client Device List Announce Request* PDU type. The RDP client sends these PDUs (one for each cached printer) upon initialization of the `RDPDR` channel. Therefore, if the victim reconnects to the server, the client will iterate on the registry keys and send them to the server, including the tampered keys with the leaks.

One way to turn the vulnerability into a *zero-click* attack is to send garbage bytes to `Microsoft::Windows::RDS::Graphics` (a dynamic channel used to transport bitmap data). Corrupting this channel shows a pop-up window that says: "connection has been lost, attempting to reconnect to the session". Then, the client effectively reconnects automatically to the server.

Finally, once the client sent the leaks, the attacker can send *Delete Printer Cachedata* PDUs to delete the leaky keys in the client's registry if they wish.

Figure 10 summarizes the attack scheme for this vulnerability.

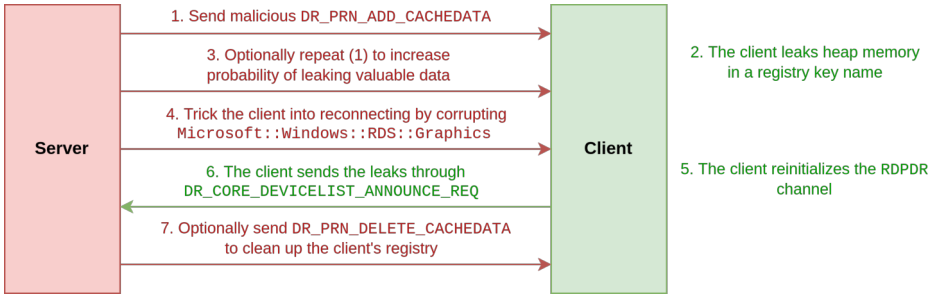


Fig. 10. Attack scheme for the Remote Heap Leak vulnerability in `RDPDR`.

Deserialization Bug / Heap Overflow in `RDPDR`. This vulnerability resides in `RDPDR`'s printer sub-protocol. It was reported to Microsoft,

which assigned it CVE-2021-38666 [21,28], and assessed it as *Remote Code Execution* of *Critical* severity.

The bug was found by analyzing crashes (which is not necessarily obvious by now). Figure 11 is a screenshot from the crash log file. Many different types of crashes occurred, across distinct modules, and even some in "unknown modules" with perplexing instruction pointer values.

One crash seemed to occur more frequently inside `RPCRT4.DLL`, thus it was the starting point for analysis. The crash arose inside the `NdrSimpleTypeConvert` function (listing 7), in the middle of what seems to be a DWORD byteswap in the heap.

```
1 mov     eax, [rdx] ; crash
2 bswap   eax
3 mov     [rdx], eax
```

Listing 7. Out-of-bounds access in `RPCRT4.DLL`.

```
1  72 44 52 49 01 00 00 00 f8 01 02 00 08 00 00 00 0e 00 00 00 00 00 00
       00 DeviceIoRequest
2  00 40 00 00 OutputBufferLength
3  00 80 2d 00 InputBufferLength
4  e8 00 09 00 IoControlCode
5  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 02 00 08 00 Padding
6  InputBuffer
7  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00
9  00 00 00 00 00 00 00 00 00 03 00 08 00 01 40 00
10 00 16 00 00 00 01 00 00 00
```

Listing 8. One of the guilty payloads.

The guilty payload (listing 8) was isolated: a *Device I/O Request* PDU, more specifically of sub-type *Device Control Request*. The `IoControlCode` field is specific to the redirected device, and the *Smart Card* sub-protocol specification [18] contains a table that maps these values to associated structure types for the input and output packets.

In parallel, the crash was successfully reproduced and analyzing the call stack leads to the function `W32SCard::LocateCardsByATRA` in `mstscax.dll`. But in fact, analyzing other payloads that trigger the same crash points out other functions (for instance, `W32SCard::WriteCache` or `W32SCard::DecodeContextAndStringCallW`). These functions all have a certain portion of code in common, shown in listing 9. Only the offset parameter (`0xE`) varies across the functions. The IOCTL table in the specification suggests there are around 60 functions of this kind.

```
Crash at time 1621526903
Exception Address: 00007ff81640d626 / 000000000005d626 (RPCRT4.dll)
Exception Information: 0000000000000000 000002513a542000

Crash at time 1621527242
Exception Address: 00007ff80a777b18 / 0000000000007b18 (WINSPOOL.DRV)
Exception Information: 0000000000000000 000002541565df98

Crash at time 1621527495
Exception Address: 00007ff8166043d2 / 00000000000743d2 (msvcrt.dll)
Exception Information: 0000000000000000 0000023744ced000

Crash at time 1621527745
Exception Address: 00007ff80a774340 / 0000000000004340 (WINSPOOL.DRV)
Exception Information: 0000000000000000 000001835bcc5ee8

Crash at time 1621527779
Exception Address: 00007ff816481bff / 00000000000d1bff (RPCRT4.dll)
Exception Information: 0000000000000000 000002d25e2f2000

Crash at time 1621527853
Exception Address: 00007fffe9a098dc / 000000001cdf98dc (unknown module)
Exception Information: 0000000000000000 00007fffe9a098dc
```

Fig. 11. Crashes while fuzzing `RDPDR`.

```
1  v6 = MesDecodeBufferHandleCreate(
2    &PDU->InputBuffer,
3    PDU->InputBufferLength,
4    &pHandle
5  );
6  // ...
7  // Crash here:
8  NdrMesTypeDecode3(
9    pHandle,
10   &pPicklingInfo,
11   &pProxyInfo,
12   (const unsigned int **)&ArrTypeOffset,
13   0xEu,
14   &pObject
15 );
```

Listing 9. Similar code pattern in several functions of the *Smart Card* extension.

The `MesDecodeBufferHandleCreate` function creates a decoding handle for RPC serialization. Indeed, RPC has its own serialization engine, called the NDR marshaling engine (*Network Data Representation*) [19], which the RDP client uses to decode structures from the PDUs.

Once the decoding handle is initialized with the input buffer, the data is effectively deserialized through `NdrMesTypeDecode3`. This function is nowhere to be documented, because developers are not supposed to use this function directly: instead, they should describe structures using Microsoft's IDL (Interface Description Language), and use the MIDL compiler to generate stubs that can encode and decode data.

The `pProxyInfo` is a `MIDL_STUBLESS_PROXY_INFO` structure that contains various informations, including the RPC interface UUID and a *Type Format String*, which is a compiled description of all the types and structures that are used within the *Smart Card* extension. The varying offset (`0xE`) then allows to select a specific structure for deserialization inside this description. Listing 10 is the structure definition for the example of the `W32SCard::LocateCardsByATRA` function.

```
typedef struct _LocateCardsByATRA_Call {
  REDIR_SCARDCONTEXT Context;
  [range(0,1000)] unsigned long cAtrs;
  [size_is(cAtrs)] LocateCards_ATRMask* rgAtrMasks;
  [range(0,10)] unsigned long cReaders;
  [size_is(cReaders)] ReaderStateA* rgReaderStates;
} LocateCardsByATRA_Call;
```

Listing 10. `LocateCardsByATRA_Call` structure in the NDR format string for the *Smart Card* extension.

To summarize the information gathered up to this point:

- The attacker can send an `IoControlCode`, an `InputBuffer` and an `InputBufferLength`.
- The input buffer is deserialized through the RPC NDR marshaling engine according to a structure that depends on `IoControlCode`.
- There are around 60 possible IOCTL calls, and thus decoding structures.
- There is an out-of-bounds read during the deserialization process, in the `NdrSimpleTypeConvert` function.

There are two key elements to the sought vulnerability. The first one is quite evident: the value of `InputBufferLength` is not properly checked, so there is a first potential overrun as the `NdrMesTypeDecode3` function may think the buffer is longer than it really is.

To understand the second one, we can take a look at the `NdrSimpleTypeConvert` function, more specifically at the moment of the crash (listing 11). The byte swap takes place when the endianness of the serialized data does not match the local endianness. Before actually decoding data, a pass on the input buffer is performed to switch the endianness of several types, in particular the `FC_ULONG` fields (which are `unsigned long` in the structure).

Therefore, in the `LocateCardsByATRA_Call` structure (listing 10), the fields `cAtrs` and `cReaders` are byte-swapped. But also and more importantly, any `unsigned long` that lies inside the nested `rgAtrMasks` or `rgReaderStates` fields will be byte-swapped. Since these fields are arrays

of structs which size is encoded inside the serialized buffer and thus controlled by the attacker, there exists a second kind of overrun. Out of all the IOCTL structures, only 3 were found to be arranged as to allow such an overrun.

```
void NdrSimpleTypeConvert(PMIDL_STUB_MESSAGE StubMsg, uchar Format)
{
  switch (Format) {
    // ...
    case FC_ULONG:
      if ((StubMsg->RpcMsg->DataRepresentation & NDR_INT_REP_MASK)
          != NDR_LOCAL_ENDIAN) {
        // Crash
        *((ulong *)StubMsg->Buffer) = RtlUlongByteSwap(*(ulong *)
            StubMsg->Buffer);
      }
      StubMsg->Buffer += 4;
    // ...
  }
}
```

Listing 11. `NdrSimpleTypeConvert` function.

By combining these two overruns, the attacker can trigger out-of-bounds operations in the heap. How does that explain the other crashes that were logged in different modules? Listing 12 shows the `LocateCards_ATRMask` structure nested inside `LocateCardsByATRA_Call`. There is an `unsigned long` field (`cbAtr`) at the beginning of this 76-bytes structure, thus an attacker may be able to byte-swap DWORDs in the heap every 76 bytes. This allows to corrupt many objects in the heap. If the input buffer length is large enough to allow out-of-bounds operations, but small enough not to exceed the heap segment, the deserialization process returns with a damaged heap.

```
typedef struct _LocateCards_ATRMask {
  [range(0, 36)] unsigned long cbAtr;
  byte rgbAtr[36];
  byte rgbMask[36];
} LocateCards_ATRMask;
```

Listing 12. `LocateCards_ATRMask` structure in the NDR format string for the *Smart Card* extension.

From there, it is suspected such behavior could be exploited to reach remote code execution, for instance by corrupting heap objects or modifying *vtable* pointers. However, we were not able to exploit this vulnerability and provide a proof of concept.

6 Conclusion

A fuzzer based on WinAFL [6] was architectured to attack Microsoft's RDP client in Windows. The *Virtual Channels* layer was targeted via the WTS API [20], opening up a large surface that was only briefly tackled through a handful of static channels. After weighing different potential strategies, some channels were effectively fuzzed and led to several bugs including CVE-2021-38665 in the *Printer* extension [22] (Important Information Disclosure) and CVE-2021-38666 in the *Smart Card* extension [21] (Critical Remote Code Execution).

Potential future work may include fuzzing other channels, fuzzing the server, or developing new fuzzing techniques, especially ones that are more adapted to channel state machines. For instance, the newer snapshot-based fuzzer *what the fuzz* [3] added support for multi-packet delivery.

Although not mentioned in this article, the fuzzer was successfully reused for fuzzing alternate client implementations of RDP, such as FreeRDP [1], which led to other CVEs (remote heap leak and arbitrary file read [8, 9]).

References

1. FreeRDP. `https://www.freerdp.com/`.
2. Andrew Swoboda, Lane Thames, Tyler Reguly. RDP Fuzzing: Why the Microsoft Open Protocol Specification is Awesome! 2014.
3. Axel "0vercl0k" Souchet. what the fuzz. 2021. `https://github.com/0vercl0k/wtf`.
4. Chun Sung Park, Yeongjin Jang, Seungjoo Kim, Ki Taek Lee. Fuzzing and Exploiting Virtual Channels in Microsoft Remote Desktop Protocol for Fun and Profit. 2019. `https://www.unexploitable.systems/papers/park:rdpfuzzing-slides.pdf`.
5. Eyal Itkin. Reverse RDP Attack: Code Execution on RDP Clients. 2019. `https://research.checkpoint.com/2019/reverse-rdp-attack-code-execution-on-rdp-clients/`.
6. Ivan Fratric. WinAFL. `https://github.com/googleprojectzero/winafl`.
7. FreeRDP. Reference Documentation. `https://github.com/FreeRDP/FreeRDP/wiki/Reference-Documentation`.
8. FreeRDP. Arbitrary file read in Windows clipboard (CVE-2021-37594). 2021. `https://github.com/FreeRDP/FreeRDP/security/advisories/GHSA-gw67-q7f9-4cg2`.
9. FreeRDP. Arbitrary file read in Windows clipboard (CVE-2021-37595). 2021. `https://github.com/FreeRDP/FreeRDP/security/advisories/GHSA-qg62-jcfp-46fw`.
10. Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018.
11. Markus Gaasedelen. Lighthouse - A Coverage Explorer for Reverse Engineers. `https://github.com/gaasedelen/lighthouse`.

12. Michal Zalewski. american fuzzy lop. https://github.com/google/AFL.
13. Microsoft. GFlags and PageHeap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap.
14. Microsoft. [MS-RDPBCGR]: Remote Desktop Protocol: Basic Connectivity and Graphics Remoting. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpbcgr/5073f4ed-1e93-45e1-b039-6e30c385867c.
15. Microsoft. [MS-RDPEA]: Remote Desktop Protocol: Audio Output Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPEA/%5bMS-RDPEA%5d.pdf.
16. Microsoft. [MS-RDPECLIP]: Remote Desktop Protocol: Clipboard Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPECLIP/%5bMS-RDPECLIP%5d.pdf.
17. Microsoft. [MS-RDPEPC]: Remote Desktop Protocol: Print Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPEPC/%5BMS-RDPEPC%5D.pdf.
18. Microsoft. [MS-RDPESC]: Remote Desktop Protocol: Smart Card Virtual Channel Extension. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-RDPESC/%5bMS-RDPESC%5d.pdf.
19. Microsoft. RPC NDR Engine (RPC). https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-ndr-engine.
20. Microsoft. wtsapi32.h header. https://docs.microsoft.com/en-us/windows/win32/api/wtsapi32/.
21. Microsoft Security Response Center. Remote Desktop Client Remote Code Execution Vulnerability (CVE-2021-38666). 2021. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-38666.
22. Microsoft Security Response Center. Remote Desktop Protocol Client Information Disclosure Vulnerability (CVE-2021-38665). 2021. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-38665.
23. Ole André V. Ravnås. Frida: A word-class dynamic instrumentation framework. https://frida.re/.
24. Shaked Reiner. Explain Like I'm 5: Remote Desktop Protocol (RDP). 2020. https://www.cyberark.com/resources/threat-research-blog/explain-like-i-m-5-remote-desktop-protocol-rdp.
25. Shaked Reiner, Or Ben-Porath. Fuzzing RDP: Holding the Stick at Both Ends. 2021. https://www.cyberark.com/resources/threat-research-blog/fuzzing-rdp-holding-the-stick-at-both-ends.
26. Stas'M. RDP Wrapper Library. https://github.com/stascorp/rdpwrap.
27. Valentino Ricotta. Remote ASLR Leak in Microsoft's RDP Client through Printer Cache Registry (CVE-2021-38665). 2021. https://thalium.github.io/blog/posts/leaking-aslr-through-rdp-printer-cache-registry/.
28. Valentino Ricotta. Remote Deserialization Bug in Microsoft's RDP Client through Smart Card Extension (CVE-2021-38666). 2021. https://thalium.github.io/blog/posts/deserialization-bug-through-rdp-smart-card-extension/.
29. WinAFL. DynamoRIO Instrumentation Mode. https://github.com/googleprojectzero/winafl/blob/master/readme_dr.md.
30. yrp. frida-drcov.py. https://github.com/gaasedelen/lighthouse/tree/master/coverage/frida.