All about the ALPC, RPC, LPC, LRPC in your PC

Alex Ionescu, Chief Architect
@aionescu
alex@crowdstrike.com

Bio

- Reverse engineered Windows kernel since 1999
- Lead kernel developer for ReactOS Project
- Interned at Apple for a few years (Core Platform Team)
- Co-author of Windows Internals 5th and 6th Edition
- Founded Winsider Seminars & Solutions Inc., to provide services and Windows Internals training
- Speaker at Recon, Blackhat, SyScan, Breakpoint, …
- Now Chief Architect at CrowdStrike
  - Security startup focused on attribution, received $50M in funding

Introduction – Pre-Vista

- Windows NT has a local inter-process communication mechanism in the kernel called LPC (Local Procedure Call)
  - Well understood, reverse engineered, several bugs found in LPC itself
  - Some LPC servers pwned at various times (and CSRSS keeps getting pwned)
- LPC is undocumented – developers use named pipes, sockets, mailslots instead for low-level data exchange
- Developers use RPC for remote procedure calls, or for more complex local function calls
  - RPC internally uses named pipes (“ncan_np” protocol)
- DCOM (Distributed COM) uses RPC internally

Introduction – Vista+

- Windows NT replaces LPC with ALPC
  - “Advanced”/”Asynchronous”
- Asynchronicity, cancellability and atomicity guarantees added to LPC make it a suitable replacement for named pipes for RPC
  - “ncalrpc” protocol is born
- Remains undocumented so not directly used by developers
  - But developers use DCOM, RPC
  - And lots of internal Windows DLLs use ALPC directly
- Result: even the simplest Windows application has at least one active ALPC connection
  - To at least one SYSTEM privileged process

The ALPC Security Reality

- Many Windows APIs are serviced through RPC/DCOM
- Windows User-Mode Drivers are DCOM objects!
- Windows Store Applications are DCOM objects!
- A bug in ALPC means:
  - Owning arbitrary ALPC servers (including the kernel)
  - Owning arbitrary RPC servers
  - Owning arbitrary DCOM objects
- A bug in LRPC means:
  - Owning arbitrary RPC servers and arbitrary DCOM objects
- And then of course there's all the individual packet processing bugs in specific ALPC servers

ALPC Basics

Message Passing, Shared Memory and Attributes

ALPC Connection Flow

- ALPC Server calls *NtAlpcCreatePort*
  - Specifies port name, attributes (such as maximum message length), and security descriptor (who is allowed to open a handle to the port)
  - Server receives an “ALPC Server Connection Port” object handle
- ALPC server calls *NtAlpcSendWaitReceivePort*
  - Can now receive incoming connection requests (LPC_CONNECTION_REQUEST)
  - Blocking call – unless asynchronous operation is used (TBD)
- ALPC client calls *NtAlpcConnectPort*
  - Specifies server port name, attributes, and an optional “connection message”
- ALPC server wakes up…

ALPC Connection Flow (2)

- … now receives the connection request from the client
  - Including any optional message packet that was sent
- Server makes decision to accept or reject connection
  - Based on client SID/state/information or the message packet
- Once decision is made, server must call *NtAlpcAcceptConnectPort*
  - BOOLEAN argument specifies acceptance or rejection
  - Server can accept connection with an optional message packet of its own
  - Server can also specify a “port context” – custom data structure that will be used to identify the client in the future
- Server now receives a handle – the “ALPC Server Communication Port”
  - Client too – the ALPC Client Communication Port

ALPC Messages

- An ALPC message is made up of a PORT_HEADER
  - Legacy LPC message header – contains message size and message type
  - Kernel fills out PID/TID of sender, and adjusts message type if needed
- Rest of the message is caller-defined and opaque
  - LPC – can be as big as 256 bytes
  - ALPC – can be as big as 64KB
- What about if the message is bigger?
  - Must use an ALPC Port Section – ALPC-managed memory manager section object (shared memory)
  - Sender and receiver receive “views” of selected portions of the section object – called “regions”
- How to specify the section view used? *ALPC Attributes*

ALPC Message Logic

- An ALPC message can be delivered in one of two ways:
  - Requests – require a reply to be sent (TCP model)
  - Datagrams – no reply can be sent (UDP model)
- If using synchronous ALPC, sender will block on the request until a reply is received
  - Or receiver blocks on reply until a new request is received
- When using asynchronous ALPC, sender can setup:
  - An I/O completion port (Windows 7 kernel, Vista+ user-mode)
  - A kernel callback object (Kernel-mode only)
  - An interlocked memory completion list (User-mode only)
- Datagrams are always inherently non-blocking since no reply is ever received

ALPC Attributes

- ALPC recognizes that certain payload data has special meaning that must be tracked in a trusted way
  - Consider this as “metadata” – similar to the Mach header and/or footer types
- For example, the memory address of the shared view – and the view itself, must accompany the message in a trusted way
  - This is called a “Data View Attribute”
- Servers may also want to receive a trusted sequence number and globally unique message ID to allow for correct management of serialized/asynchronous processing
  - This is called a “Context Attribute”

ALPC Attributes (2)

- Servers may want to know the security context of the client at the time the message was sent
  - This is called the "Security Attribute"
- And servers may want to receive handles to objects in a safe way
  - Encoded in a "Handle Attribute"
- Each time a message is sent, output attributes can be specified, telling ALPC what metadata to send with the payload
  - Output attributes are validated and "captured" in the kernel
- Each time a message is received, ALPC input attributes can be specified, telling ALPC what metadata to "expose" with the payload

ALPC Attributes Logic

- A sender can always attempt to send valid attributes…
- But if the receiver does not specify interest in the attributes, the call can fail
  - Or the server will simply not have any of the attributes “exposed”
- For the security and context attribute, the data is hosted by the kernel itself – exposing this data merely means returning it to the server
- But for the data view attribute, exposing it means mapping the view in the server’s address space
- And for the handle attribute, exposing it means duplicating the handle in the server’s object table
- If a server accepts data view or handle attributes..
  - It must free the view or close the handle as needed

Handle Attribute

- Handle attributes enable safe exchange of handles
- Sender:
  - Specifies which type of object it's sending
  - Passes in local handle
- When sending the message, ALPC:
  - Validates a handle is present
  - Validates the handle matches the type of object caller specified
  - Makes a kernel copy of the handle (duplicated in System)
- Receiver:
  - Specifies which type of object it's expecting
- When receiving the message, ALPC:
  - Validates the type of object matches what sender sent
  - Makes a local copy of the handle (duplicated in local process)

View Attribute

- View attributes enable passing data over 64KB
- View attribute can be made auto-release
  - ALPC runtime will dynamically unmap the view as needed
- View attribute can also be made secured
  - Sender view will be made Read-Only after message is sent
  - Receiver view will be made Read-Write after message is received
  - *MmSecureVirtualMemoryAgainstWrites*, *MmSecureVirtualMemory*, SEC_NO_CHANGE are used to prevent unmapping, remapping, protection changing, and section changing by the sender
  - Secured views are a key component of Distributed COM (DCOM)
- Sender creates the view attribute, receiver gets mapping

Setting Port Attributes

```
37  VOID
38  InitializeAlpcClientPort (
39      _Out_ PALPC_PORT_ATTRIBUTES PortAttributes
40      )
41  {
42      //
43      // Initialize a port with 60KB messages and dynamic security tracking
44      //
45      RtlZeroMemory(PortAttributes, sizeof(*PortAttributes));
46      PortAttributes->MaxMessageLength = 60 * 1024;
47      PortAttributes->SecurityQos.Length = sizeof(SECURITY_QUALITY_OF_SERVICE);
48      PortAttributes->SecurityQos.ContextTrackingMode = SECURITY_DYNAMIC_TRACKING;
49      PortAttributes->SecurityQos.EffectiveOnly = TRUE;
50      PortAttributes->SecurityQos.ImpersonationLevel = SecurityAnonymous;
51  }
```

Initializing a Message

```
23  VOID
24  InitializeConnectMessage (
25      _Out_ PMSG_CONNECT MsgConnect
26      )
27  {
28      //
29      // Initialize the connection message this server needs.
30      // An empty message is ok.
31      //
32      RtlZeroMemory(MsgConnect, sizeof(*MsgConnect));
33      MsgConnect->h.u1.s1.TotalLength = sizeof(*MsgConnect);
34      MsgConnect->h.u1.s1.DataLength = sizeof(*MsgConnect) - sizeof(PORT_MESSAGE);
35  }
```

Connecting to a Server

```
143     //
144     // Connect to the server port, passing in the connection message
145     //
146     bufferSize = sizeof(connectMessage);
147     status = NtAlpcConnectPort(&serverPort,
148                                &serverName,
149                                NULL,
150                                &portAttributes,
151                                ALPC_MSGFLG_SYNC_REQUEST,
152                                NULL,
153                                &connectMessage.h,
154                                &bufferSize,
155                                NULL,
156                                NULL,
157                                NULL);
158     if (!NT_SUCCESS(status))
159     {
160         printf("Failed to connect to server: %lx!\n", status);
161         return;
162     }
163     else
164     {
165         printf("Connected to server! Handle: %lx\n", serverPort);
166     }
167
168     //
169     // Initialize the message attributes. We just need to do this once, we'll
170     // re-initialize the fields as needed each loop
171     //
172     status = AlpcInitializeMessageAttribute(ALPC_FLG_MSG_DATA_VIEW_ATTR,
173                                             msgAttributes,
174                                             sizeof(attributesBuffer),
175                                             &bufferSize);
176     if (!NT_SUCCESS(status))
177     {
178         printf("Failed to initialize data view attribute: %lx!\n", status);
179         return;
180     }
```

Creating a View

```
62      HANDLE sectionHandle;
63      NTSTATUS status;
64      PALPC_DATA_VIEW_ATTR dataView;
65
66      //
67      // Assume failure
68      //
69      *SectionHandle = NULL;
70      *DataView = NULL;
71
72      //
73      // Create a 4KB section for the port, with a 4KB view inside of it
74      //
75      status = NtAlpcCreatePortSection(ServerPort, 0, NULL, *ViewSize, &sectionHandle, ViewSize);
76      if (!NT_SUCCESS(status)) return status;
77
78      //
79      // Setup a 4KB auto-release view on the client side
80      //
81      dataView = AlpcGetMessageAttribute(Attributes,
82                                         ALPC_FLG_MSG_DATA_VIEW_ATTR);
83      dataView->SectionHandle = sectionHandle;
84      dataView->ViewBase = NULL;
85      dataView->ViewSize = 1 * PAGE_SIZE;
86      dataView->Flags = 0;
87      status = NtAlpcCreateSectionView(ServerPort, 0, dataView);
88      if (!NT_SUCCESS(status))
89      {
90          //
91          // We failed, destroy the section we just created
92          //
93          NtAlpcDeletePortSection(ServerPort, 0, sectionHandle);
94      }
95      else
96      {
97          //
98          // Mark the view as autorelease
99          //
100         dataView->Flags = ALPC_VIEWFLG_AUTORELEASE;
101
102         //
103         // Return the handle and view to the caller
104         //
105         *SectionHandle = sectionHandle;
106         *DataView = dataView;
107     }
108
109     //
110     // Return the result back
111     //
112     return status;
```

Sending a Message

```
206     //
207     // Fill the view with our magic value and set the attribute as active
208     //
209     msgAttributes->ValidAttributes |= ALPC_FLG_MSG_DATA_VIEW_ATTR;
210     RtlFillMemory(dataView->ViewBase, PAGE_SIZE, 0x41414141);
211
212     //
213     // Send an empty message to the server, but attached to an attribute
214     //
215     RtlZeroMemory(hackMessage, sizeof(msgBuffer));
216     hackMessage->u1.s1.TotalLength = sizeof(msgBuffer);
217     hackMessage->u1.s1.DataLength = hackMessage->u1.s1.TotalLength -
218                                     sizeof(PORT_MESSAGE);
219     bufferSize = sizeof(msgBuffer);
220     status = NtAlpcSendWaitReceivePort(serverPort,
221                                        ALPC_MSGFLG_RELEASE_MESSAGE,
222                                        hackMessage,
223                                        msgAttributes,
224                                        NULL,
225                                        &bufferSize,
226                                        NULL,
227                                        NULL);
228     if (!NT_SUCCESS(status))
229     {
230         printf("Failed to send message: %lx!\n", status);
231         return;
232     }
```

ALPC Heap-Spray

Resource Exhaustion through Data View and Handle Attributes

Data View Release

- If the receiver has a data view exposed, it is mapped in the address space
- The receive has two options to mark the view for release
  - Set ALPC_VIEWFLG_UNMAP_EXISTING in the view attribute
  - Manually call the *NtAlpcFreeSectionView* API
- But the view is still active at this point, because it is still bound to the message!
- Receiver must:
  - Set the ALPC_MSGFLG_RELEASE_MESSAGE flag in the reply
  - Send the reply to the client
- If a server doesn’t expect to receive a view and ignores it – it will be leaked
- If a server doesn’t reply to a message – it will be leaked

Replying to Messages

- An ALPC server developer can make several understandable incorrect assumptions
- “This message should never have a view…”
  - Let’s not check for one
  - If I get one, let me drop the message on the floor or cancel it
  - If I get one, let me free the view/mark the view for unmapping
  - If I get one, let me free the view/mark the view for unmapping, and reply to the server if this a request (datagrams don’t need replies)
  - If I get one, let me free the view/mark the view for unmapping, and reply to the server while setting the ALPC_MSGFLG_RELEASE_MESSAGE flag (datagrams don’t need replies)
- All of these will leak the view!

Continuation… REQUIRED!

- The final incorrect assumption (in the last bullet) is that datagrams don't need replies
- This is true in the strict protocol sense – the sender is not expecting, nor waiting for a reply to a datagram
  - And ALPC will never actually send the reply
- But the whole ALPC attribute release logic hinges upon the receiver indicating message processing completion by "replying" with the ALPC_MSGFLG_RELEASE_MESSAGE flag
  - This reply is thus an internal ALPC semantic state flow
- Many ALPC servers don't reply to datagrams – ever
  - But ALPC warns you when this is wrong – message header has LPC_CONTINUATION_REQUIRED flag set

Results

- Identified several Windows 7 ALPC servers that either:
  - Leak unexpected data view attributes
  - Leak unexpected handle attributes
- ALPC servers can be subjected to a Denial-of-Service attach by preventing them from responding to further ALPC messages and exhaust their address space
- Can also be used as a heap-spraying technique
  - List of vulnerable servers includes several SYSTEM-level services
- Identified an additional X Windows 8.1 ALPC servers that suffer from the same issue
- Currently preparing security vulnerability report for MSRC

Local RPC (LRPC)

Internal LRPC Marshalling Over ALPC

ncalrpc

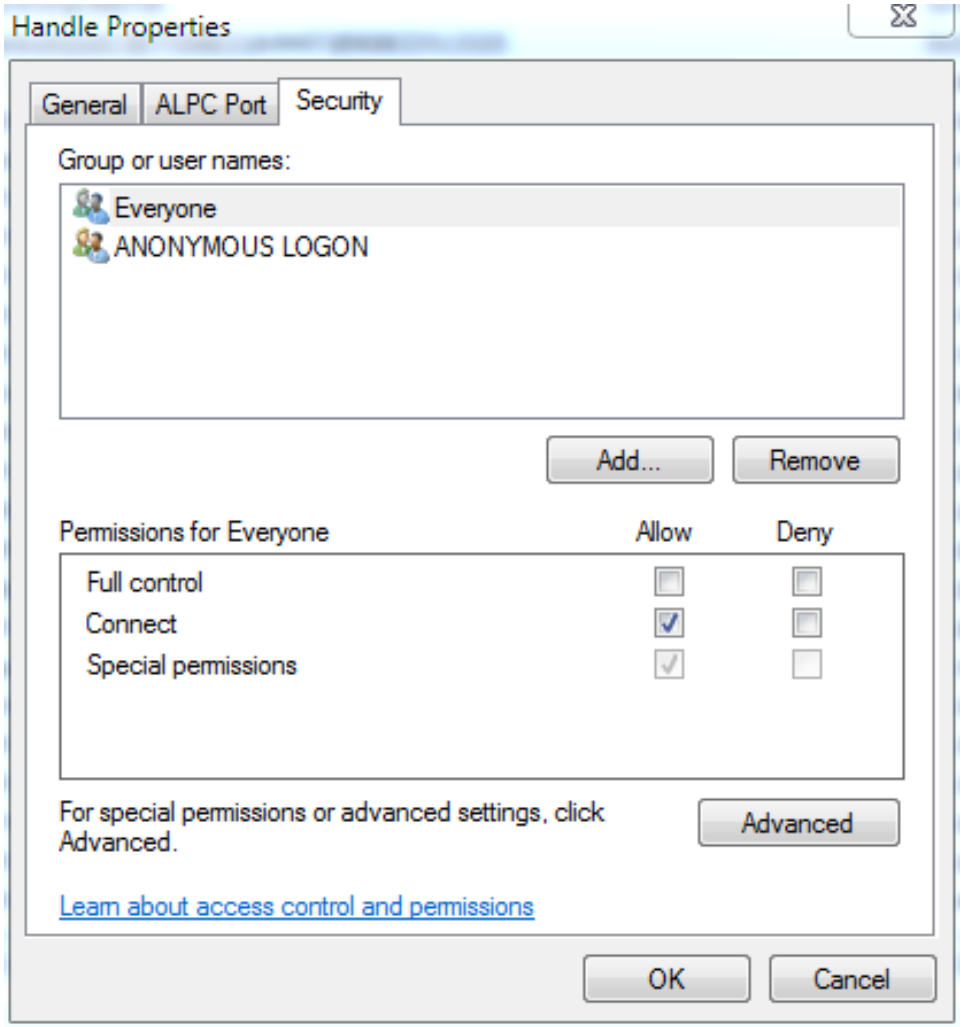
- Starting with Vista, the RPC Runtime (rpcrt4.dll) now supports using ALPC as the backing protocol for local RPC
  - Navigate to \RPC Control in the Object Namespace and you'll see hundreds of ports
- In Windows 7, the Kernel RPC Runtime (msrpc.sys) also allows talking to RPC servers in user-mode by using ALPC
- Extensions in WinDBG such as !alpc /P, /lpc, /lpp can be useful to analyze connections between clients and servers
- But the ALPC layer only tells one side of the story – let's look at how LRPC works under the hood

LRPC

- RPC servers can specify which RPC clients they want to allow a connection from, as well as specific binding semantics
  - Making sure that the right IID (Interface ID/GUID) is used, the right version, the right marshalling syntax (NDR, NDR 2.0, NDR 64/3.0), etc
- All these checks are done by the RPC runtime as part of a “bind” request
  - Bind requests are not sent at connection time – they are bonafide ALPC request packets
  - In other words, an ALPC connection has already been established
- Indeed, looking at the security descriptor of any ALPC port…

Let them connect

Everyone is welcome!



LRPC Semantics

- As soon as an RPC server is registered, the RPC runtime creates a named port in \RPC Control
  - .idl file can actually override this!
- RPC runtime starts a server loop that waits for ALPC messages
- RPC runtime requests security, data view, and context attributes for all messages
  - Potential exists for “unexpected” attributes to be leaked
- Complex loop (all in one function!) accepts and closes connections, requests, and error packets
  - LRPC_ADDRESS::ProcessIo

LRPC Packet Types

- LRPC uses tagLrpcMessageTypes enumeration:
  - Request (0) & Bind (1) are the primary packet types used
  - Fault (2) used to send protocol errors
  - Response (3) & Cancel (4) used in Asynchronous Calls
  - Callbacks (6-11) used in Callback-based RPC
  - Pipes (12 and 13) used in Pipe-based RPC
- All LRPC messages have a LRPC_PACKET_HEADER:
  - Standard PORT_MESSAGE from LPC and type from above
- LRPC_SHORT_CTRL_MESSAGE is the basic packet types
  - LRPC_REQUEST_MESSAGE, LRPC_RESPONSE_MESSAGE, LRPC_PIPE_ACK inherit from this
- LRPC_BIND_MESSAGE is separate (and a few others)

LRPC Fuzzing

- The RPC runtime has pretty good management of LRPC packets and state transitions
- Checked build contains a number of ASSERTions to catch protocol errors
  - Retail build will send Fault packets with RPC_S_PROTOCOL_ERROR
- Was unable to get any interesting behaviors with arbitrary/random/out-of-order LRPC packets
- But I'm not Ben Nagy ;-)
- Wrote a simple lrpcfuzz.exe which dumps all LRPC ports, tries to connect with them at the ALPC level, and then sends arbitrary packets and malformed LRPC packets

LRPC Data View Unmapping

- So, started taking a look at data view attribute freeing…
- RPC runtime does this so well, it should've been used as a template for other ALPC servers!
  - Always checks if data view attribute is present
    - ASSERTs that LPC_CONTINUATION_REQUIRED is set
    - Sets an internal "reply" flag as a local variable
  - Always builds an attribute with ALPC_VIEWFLG_UNMAP_EXISTING set
- Always replies if "reply" local variable is set
- In fact, in some cases, runtime even over-aggressively drops views multiple times (this is fine)
- RPC runtime doesn't use handle attributes, so no bugs there...

LRPC Heap Spray

- Each time my fuzzer hit a potential path, it was met with [redacted]
- Noticed something interesting: this function doesn't actually free the message buffer
  - [redacted]
- RPC runtime was doing this correctly in all cases… except one!
- So my fuzzer found one case in which [redacted] the actual packet was being left in memory
- Packet can be made as big as 64K and thus slowly fill up the entire address space of any local RPC server!

Results

- Wrote “lrpcnuke.exe” and tried hitting various RPC servers
- Achieved DoS 100% of the time – in some cases, killing the entire machine since the service is critical and/or deadlocks
- Was able to reliably fill address space with controlled values (41414141 sled)
  - Could be combined with other vulnerability as a heap-spray
- Did not yet find any server that crashed (but tried less than a dozen)
  - A crashing server could be bad handling of out-of-memory issue
  - Could potentially result in exploitable path
- Reported to MSRC in early October (180 days ago)

ANALYSIS TIME

ALPC and LRPC in WinDBG

DEMO TIME

Killing ALPC Server and RPC Servers

Conclusion

- ALPC is really complex
  - ALPC Resource Management gives you free atomicity, security, and consistency…
  - … at the cost of knowing how to use it!
- Microsoft developers should probably have an internal ‘brown bag’ and design document on how to correctly use ALPC
- The LRPC bug is a simple mistake
  - Probably best not to have one single 4KB function handling all possible code paths
  - LRPC runtime is actually extremely well written ALPC engine
- Additional ALPC/LRPC fuzzing needed
  - Also looking into more DCOM internals
- Greetz to Ben Nagy, Thomas Lim, Loren Robinson