This next series of posts will focus on looking at a game's protocol. Specifically, the packet structure and some basic steps into the world of packet reversing. I will focus on explaining how a specific packet (a chat packet) is built in-game and how understanding the packet structure can let someone do some fun and unexpected things. It should be noted that this won't be an article in server emulation as that is much too complex and time consuming to both perform and write about. The steps and methodologies discussed though can aid in beginning to fully reverse a protocol and begin a project in server emulation. This whole series will be broken down into three parts:

- A preliminary post (this one) discussing how to perform a high level inspection of a protocol.
- Reverse engineering the target application to see where and how a specific packet is built.
- Example programs that could be written to send your own packets or modify parts of incoming/outgoing packets.

To start off, the target will be Age of Empires II: The Conquerors Expansion. This was chosen because the chat packets of the protocol are unencrypted so there won't be any large sidetrack about hunting down encryption keys or reversing algorithms. There are other factors like the game being dead (developing studio disbanded), and it being one of the few multiplayer games that I own. That said, it's time for some analysis. One of the better tools for this job is WPE Pro. It's extremely easy to use and provides a simple interface that displays everything that is needed. To get started, I created a LAN game with myself and used WPE to log packets on one instance. Below is an example of some chat packets that were sent. These can be analyzed for patterns and guesses at the different parts of it can be taken and later verified or discarded. The hex dump of the packets is shown below, with the sent chat in blue and the changes between the packets highlighted in red:

```
D8 26 D9 16 00 00 00 00 43 17 00 00 06 00 00 00 E7 07 00 00 01 DC 59 59 4E 4E
4E 4E 4E 4E 32 00 01 00 00 00 01 41 00 00 24 DC 18
D8 26 D9 16 00 00 00 00 43 18 23 02 06 00 00 00 E8 07 00 00 01 DC 59 59 4E 4E
4E 4E 4E 4E 32 00 02 00 00 00 01 41 41 00 24 DC 18 00
D8 26 D9 16 00 00 00 00 43 19 18 00 06 00 00 00 E9 07 00 00 01 DC 59 59 4E 4E
4E 4E 4E 4E 32 00 03 00 00 00 01 41 41 41 00 DC 18 00 80
D8 26 D9 16 00 00 00 00 43 1A 18 00 06 00 00 00 EA 07 00 00 01 DC 59 59 4E 4E
4E 4E 4E 4E 32 00 04 00 00 00 01 41 41 41 41 00 18 00 80 DC
D8 26 D9 16 00 00 00 00 43 1B 23 02 06 00 00 00 EB 07 00 00 01 DC 59 59 4E 4E
4E 4E 4E 4E 32 00 05 00 00 00 01 41 41 41 41 41 00 00 80 DC 54
```

These packets show sending "A", "AA", "AAA", "AAAA", and "AAAAA" to the chat, which can be seen (0×41 characters). One thing to immediately note is that the name of the player sending the chat is not contained in the packet. That means that there is some field in the packet which denotes who is sending the packet. There also appear to be a few flags that serve as counters. The first packet contains 0×17 and 0xE7, the next one increments these bytes to 0×18 and 0xE8, the one afterwards to 0×19 and 0xE9, and so on. There is also a lone field in the packet which matches the size of the chat being sent. Lastly, there are some unknown values which seem to change with each packet sent. To get a better understanding of things, it is better to log packets over multiple instances/connections. The game was restarted and the same five packets were sent. Reproduced below are the five packets with their differences and chat highlighted. Additionally, the logging was done from sending chat as the second player in the game session.

```
DD BF 35 17 00 00 00 00 43 07 00 00 06 00 00 00 A7 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 01 00 00 00 01 41 00 00 24 DC 18
DD BF 35 17 00 00 00 00 43 08 3D 02 06 00 00 00 A8 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 02 00 00 00 01 41 41 00 24 DC 18 00
DD BF 35 17 00 00 00 00 43 09 18 00 06 00 00 00 A9 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 03 00 00 00 01 41 41 41 00 DC 18 00 80
DD BF 35 17 00 00 00 00 43 0A 18 00 06 00 00 00 AA 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 04 00 00 00 01 41 41 41 41 00 18 00 80 DC
DD BF 35 17 00 00 00 00 43 0B 3D 02 06 00 00 00 AB 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 05 00 00 00 01 41 41 41 41 41 00 00 80 DC 54
```

Changes appeared to have happened within the expected offsets in the packets. Comparing differences between two packets from two different sessions:

```
D8 26 D9 16 00 00 00 00 43 1B 23 02 06 00 00 00 EB 07 00 00 01 DC 59 59 4E 4E
4E 4E 4E 4E 32 00 05 00 00 00 01 41 41 41 41 41 00 00 80 DC 54
DD BF 35 17 00 00 00 00 43 0B 3D 02 06 00 00 00 AB 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 05 00 00 00 01 41 41 41 41 41 00 00 80 DC 54
```

The 0x1B and 0x0B flags are highlighted differently because they're supposed to be the same. There were some intermediate message sent between testing in one of the sessions and it threw off the counter. Everything else should be normal. One of the "unexpected" changes was that 0×01 changed to 0×02 and 0×32 changed to 0x4B. Since the only change aside from a new session was that the sending player was in the second player slot, it might be reasonable to guess that the byte that changed from 0×01 to 0×02 holds the number of the player who is sending the chat. The change in the other byte is unknown. The other unexpected change was that the first four bytes of the packet also changed. For the sake of not having to explain, I'll simply mention that the first four, possibly eight, bytes appear to be a session key since they precede all packets sent throughout the game (not just chat ones). The actual value of the key is calculated external to the game, in the DirectPlay networking library. Finding out how the key is negotiated is still something that I am working on and won't be covered in this series of posts — perhaps at a future date as my progress develops. The downside to not knowing how to calculate this is that in order to send custom packets to the client the connection must already exist. Obtaining the key is not really a problem however as sendto/recvfrom can be hooked and the key stored. The next thing to note is about the last four bytes of the packet. These are also external to game and appended by DirectPlay. From my testing, these four bytes appear to be a checksum on the size of the packet. For example, sending "a", "b", "q", "$", or any one character chat text will append 00 24 DC 18 to the packet. All two character texts have the same bytes appended at the end, and so on. The last thing is about the eight bytes after the player index. In the samples these bytes 0×59 and 0x4E. For a high level analysis, it is important to test as many things as possible. The in-game chat has three options: all (default), team, and enemy. Testing sending messages using these three options reveals that these eight bytes are the "audience" flags of who of the at most eight players in a game can see the chat message. Given that a majority of the packet has been understood and guessed at, this completes the high level preliminary analysis. A sample packet is shown below and guessed at or known fields are highlighted.

```
DD BF 35 17 00 00 00 00 43 0B 3D 02 06 00 00 00 AB 0F 00 00 02 DC 59 59 4E 4E
4E 4E 4E 4E 4B 00 05 00 00 00 01 41 41 41 41 41 00 00 80 DC 54
```

Red indicates a session key.
Blue indicates some sort of counter bytes.
Pink indicates the player sending the chat.
Green indicates who is to receive the message.

Yellow indicates the length of the chat string.
Grey indicates the chat string and the null terminator.
Teal indicates the trailing checksum.
Orange indicates unknown bytes that were constant across multiple sessions.
Black indicates unknown bytes that were variable across multiple sessions.

One last point to make is that at a high level, how some of these bytes are interpreted could be off. For example, the size of the packet shown as 0×05 could actually be given two bytes, 0×00 and 0×05 in the packet, making that orange unknown byte part of the size. The 0×06, 0×00, 0×00, 0×00 bytes could actually be a DWORD instead of four unique, unrelated properties, and so on. The high level analysis is good for getting started but the actual application will require reverse engineering to fully know what is going on. This will be the topic of the next post.