

So when making a TCP connection, as Vulnserver does, the server has to set up the socket in this manner. The binding and listening occurs on a per-port basis, which is why this exploit is a socket *reuse*, and not just making a new socket, because that would mean opening up a new port. Once the socket and connection is established, programs can call send() or recv() to exchange data. Recv() will accept incoming data and write it somewhere to memory. The goal of this exploit is to invoke the recv() syscall, pass in the proper parameters to reuse Vulnserver’s socket, and then send a second-stage payload that won’t get truncated because it doesn’t need to overflow the KSTET command (which is what I suspect is truncating the first payload).

Before any connection is made from either the server or client, a socket is first created.

A socket can then either beL

* passed to a listen function (indicating that it should listen for new connections and accept them),
* passed to a connect function (indicating it should connect to another socket that is listening elsewhere)

As a socket represents a connection to another host, if you have access to it - you can freely call the corresponding send or recv functions to perform network operations.

This is the end goal of a socket reuse exploit.

By identifying the location of a socket, it is possible to listen for more data using the recv function and dump it into an area of memory that it can then be executed from - all with only a handful of instructions that should fit into even small payload spaces.

You may be asking - why not just create a new socket? The reason for this, is that a socket is bound to a port - meaning you are not able to create a new socket on a port that is already in use.

If you were to create a socket listening on a different port altogether, it would lose reliability given most targets would typically be behind a firewall.

Analysis and Socket Hunting

The first thing we need to do before we can start putting together any code is to figure out where we can find the socket that the data our exploit is sending is being received on.

If you recall from the earlier section of this post, the function calls follow the pattern of socket() > listen() > accept() > recv() if a server is accepting incoming connections and then receiving data from the client.

With this in mind, we should restart the application and let it pause at the entry point (the second breakpoint that is automatically added) and begin to search for these system calls.

As the Vulnserver application is quite simple, we don’t have to search very far. By scrolling down through the instructions we can find the point at which the welcome message is sent to the client (which is sent after the client connects) and the subsequent call to recv that precedes the processing of the command sent by the end user:

A screenshot of a cell phone

Description automatically generated

If we now place a breakpoint on the call <JMP.&recv> instruction and resume execution, we will be able to inspect the arguments that are being passed to the function on the stack.

A screenshot of a social media post

Description automatically generated

Without any context, these values will make no sense. Thankfully, detailed documentation of these functions is provided by Microsoft. In this case, we can find the documentation of the recv function at <https://docs.microsoft.com/en-us/windows/desktop/api/winsock/nf-winsock-recv>.

As can be seen in the documentation, the signature of the recv function is:

**int** **recv**(

SOCKET s,

**char** **\***buf,

**int** len,

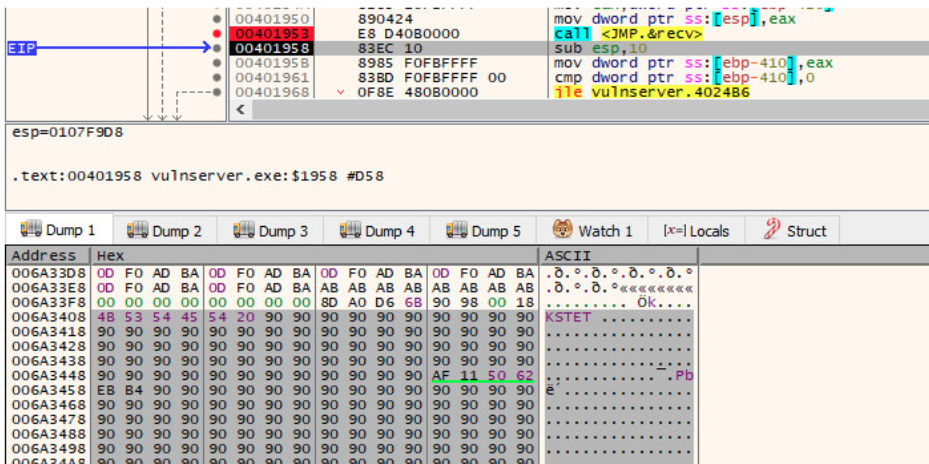
**int** flags

);

This now allows us to make sense of the arguments that we can see sat on the stack.

* The first argument (on the top of the stack) is the socket file descriptor; in this, case the value 0x128.
* The second argument is the buffer, i.e. a pointer to the area of memory that the data received via the socket will be stored. In this case, it will store the received data at 0x006a3408
* The third argument is the amount of data to expect. This has been set at 0x1000 bytes (4096 bytes)
* The final argument is the flags that influence the behaviour of the function. As the default behaviour is being used, this is set to 0

If we now step **over** the call to recv, and then jump to 0x006a3408 in the dump tab, we will see the full payload that was sent by the exploit:



With an understanding of the function call now in hand, we need to figure out how we can find that socket descriptor once more to use in our own call to recv. As this value can change every time a new connection is made, it cannot be hard coded (that would be too easy!).

Before moving on, be sure to double click the call instruction and make note of the address that recv is found at; we will need this later. In this case, it can be found at 0x0040252C.

If we now allow the program to execute until we reach our NOP sled once more, we will run into a problem. When we look at where the file descriptor was initially on the stack when the program called recv (i.e. 0x0107f9c8), it is no longer there - our overflow has overwritten it!

Although our buffer reaches just far enough to overwrite the arguments that are passed to recv, the file descriptor will still exist somewhere in memory. If we restart the program and pause at the call to recv again, we can start to analyse how it finds the file descriptor in a bit more depth.

As the socket is the first argument passed to recv / the last argument to be pushed on to the stack, we need to find the last operation to place something on the stack before the call to recv. Conveniently, this appears directly above the call instruction and is a mov that moves the value stored in $eax to the address pointed to by $esp (i.e. the top of the stack).