Task 1 - Fuzzing with Boofuzz - SpiderCat

The SpiderCat.exe program is hardcoded to bind a listener on TCP port 27015. It accepts four (4) inputs; HELP, BABY, CAT, and SPIDER.

The Boofuzz session option reuse_target_connection=True needs to be used with SpiderCat.exe so Boofuzz doesn't close the connection for each test case because SpiderCat will exit upon the first connection closing the connection.

```
session = Session (
    target=Target(
        connection=SocketConnection(host,port, proto='tcp'),
        # monitors=monitors
    ),
    reuse_target_connection=True,
    sleep_time=1,
    fuzz_loggers=[FuzzLoggerText(),FuzzLoggerCsv(file_handle=csv_log)]
    )
```

Some things that apply globally to this program:

The recv() buffer named buff is 260h or 608 bytes in size.

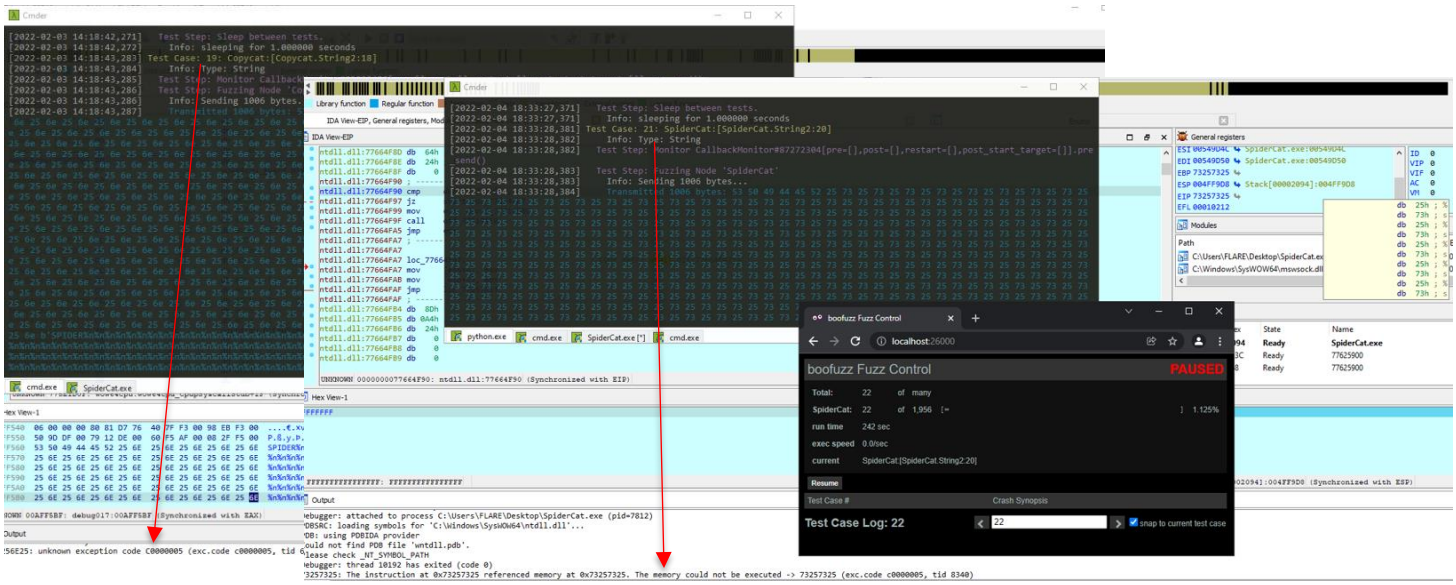
```
; int __cdecl main(int argc,
_main proc near

WSAData= WSAData ptr -3F0h
buf= byte ptr -260h
pHints= ADDRINFOA ptr -60h
```

Fuzzing the SPIDER command

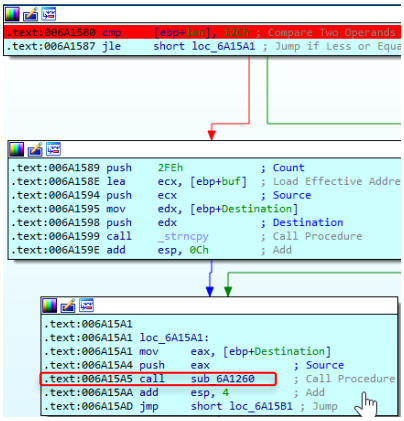
The following options were used with Boofuzz.

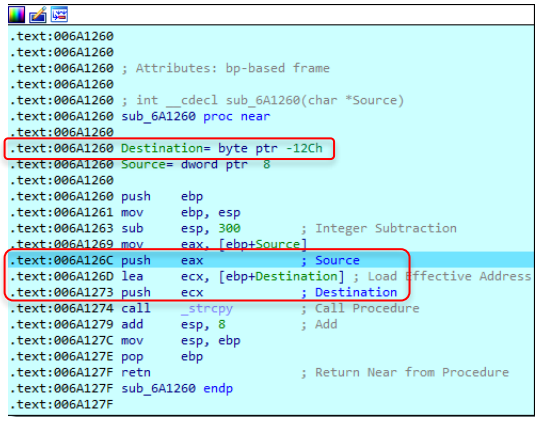
```
s_initialize("SpiderCat")
s_string("SPIDER", fuzzable=False)
s_string("FUZZ")

session.connect(s_get("SpiderCat"))
session.fuzz()
```

SpiderCat.exe crashed quickly and it ended up being test case pattern 19 and 21 crashed it. I didn't keep doing this because as you will see below, anything over 300 bytes is going to cause a crash for this command path. I utilized the boofuzz session option index_start=20 to start fuzzing from there so I could get past 19.



If the recv() buffer length is greater than 300 bytes it jumps to a block with a custom function, in this case is sub_6A1260, which is where the vulnerability issue is. In this function it copies what was in the original recv() buffer into the Destination variable with strcpy(). The Destination buffer is 12Ch/300 bytes. Therefore, anything that is more than 300 bytes causes an overflow.

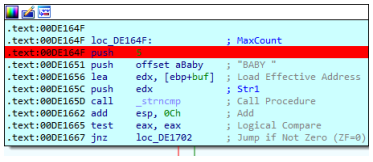




```
.text:006A1260
.text:006A1260
.text:006A1260 ; Attributes: bp-based frame
.text:006A1260
.text:006A1260 ; int __cdecl sub_6A1260(char *Source)
.text:006A1260 sub_6A1260 proc near
.text:006A1260
.text:006A1260 Destination= byte ptr -12Ch
.text:006A1260 Source= dword ptr  8
.text:006A1260
.text:006A1260 push    ebp
.text:006A1261 mov     ebp, esp
.text:006A1263 sub     esp, 300        ; Integer Subtraction
.text:006A1269 mov     eax, [ebp+Source]
.text:006A126C push    eax             ; Source
.text:006A126D lea     ecx, [ebp+Destination] ; Load Effective Address
.text:006A1273 push    ecx             ; Destination
.text:006A1274 call    _strcpy         ; Call Procedure
.text:006A1279 add     esp, 8          ; Add
.text:006A127C mov     esp, ebp
.text:006A127E pop     ebp
.text:006A127F retn                    ; Return Near from Procedure
.text:006A127F sub_6A1260 endp
.text:006A127F
```

Fuzzing the BABY command

This command requires that you send “BABY ~” to SpiderCat.exe to have it take the proper path. It does this by pushing 5 (max length) and “BABY “ to the stack and compares that with the input sent in Str1/buff[].

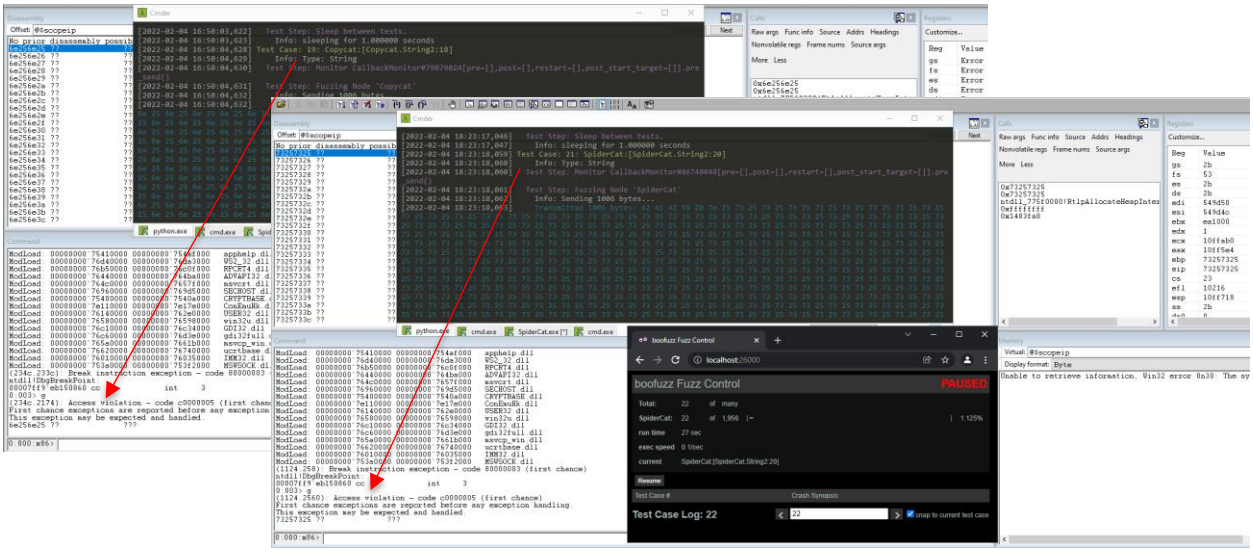


```
mov     ecx, [ebp+var_24]
movsx   edx, [ebp+ecx+buf]
cmp     edx, 7Eh ; '~'
jnz     short loc_5316E9
```

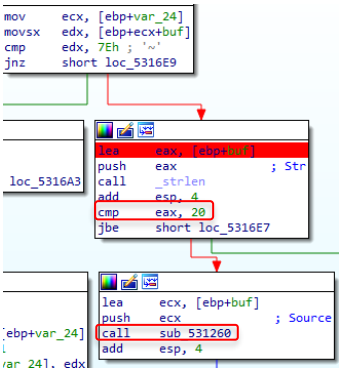
Boofuzz options for BABY:

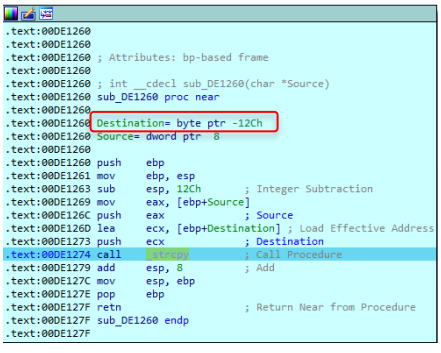
```
s_initialize("SpiderCat")
s_string("BABY ~", fuzzable=False)
s_string("FUZZ")

session.connect(s_get("SpiderCat"))
session.fuzz()
```

Boofuzz Test Case 19 and 21 caused a crash. I didn't keep doing this because as you will see below, anything over 300 bytes is going to cause a crash for this command path. I utilized the boofuzz session option index_start=20 to start fuzzing from there so I could get past 19.



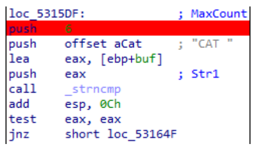
Anlyzing this further in IDAPro, after you pass "BABY " to SpiderCat it enters the BABY routine and first checks the end of the string for the tilde (~) character or if the tilde happens to be the first char after the space in "BABY ". If the tilde (~) character is there it jumps to a block that gets the total length of the string and if it's longer than 14h/20 bytes it jumps to the next block where there is a custom function named, in this case, sub_DE1260. That function is where the strcpy() is. This is the same vulnerable function as the one that the SPIDER command calls. It is named differently because I closed IDA at some point in between testing and blew away the DB. The purpose of this function is to move the source string into the destination address. The Destination location is 12Ch or 300 bytes in size. Anything larger than 300 bytes will crash the program.





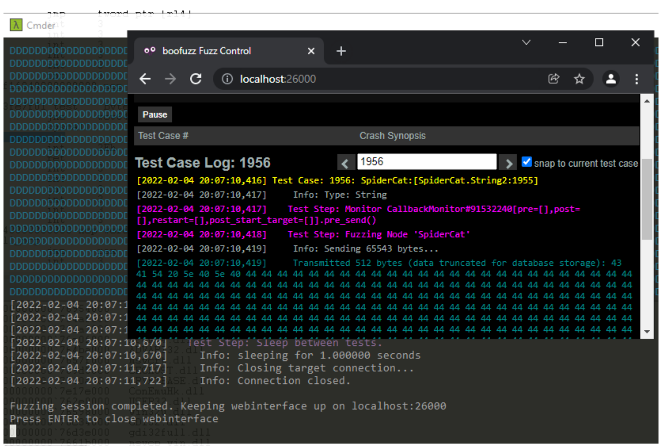
Fuzzing the CAT command

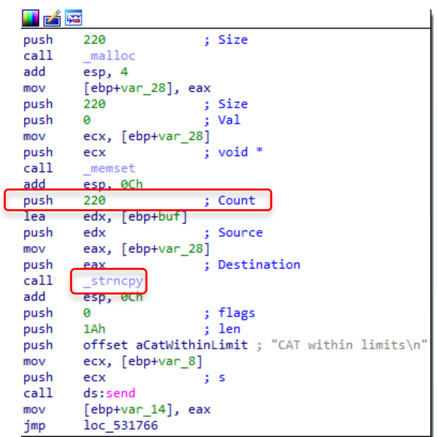
Like the others you need to get the program to first take the correct path to fuzz this command. This one pushes 6 and "CAT " to the stack before doing the strncmp() with the string you give it. The issue is that there could be residual data in memory at that point after the ' ' (space/0x20h) in the string and due to the length of 6 bytes comparison you need to pad the sent string with null bytes. You can do this with the '^@' character. So, setting up the boofuzz script you make this string unfuzzable with the padding for the program to take the proper path.




```
s_initialize("Copycat")
s_string("CAT ^@^@", fuzzable=False)
s_string("FUZZ")

session.connect(s_get("Copycat"))
session.fuzz()
```

BooFuzz ran through all test cases and did not crash the program utilizing the CAT command. The reason for this is that the strcpy() funtion in this case is not strcpy but strncpy(). This function allows for a 3rd parameter, which is used in this case, for max length to be passed to the function to help prevent an overflow condition. In this case it allows only 220 bytes in this case to be copied to the destination no matter how much is sent.





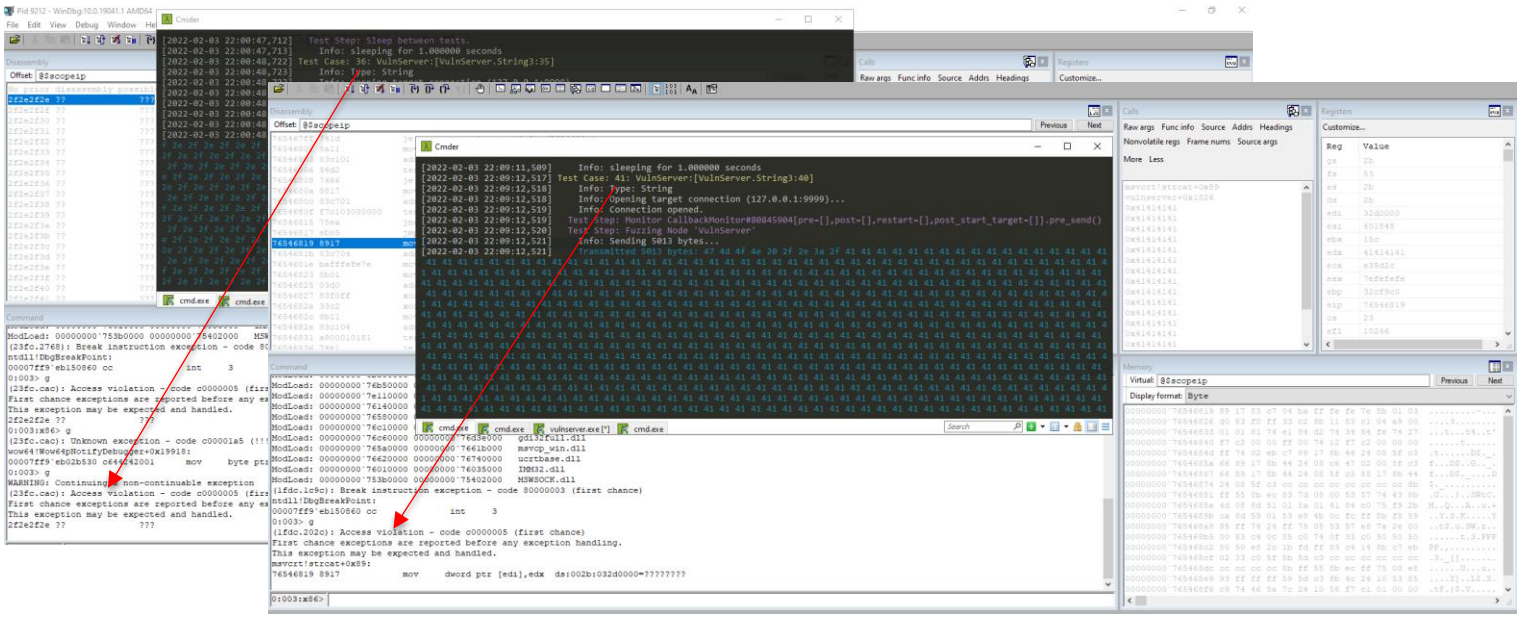
Task 2 - Fuzzing Vulnserver with Boofuzz

Fuzzing the GMON Option

Boofuzz options. You need the '\' to take the proper path. There are a few Boofuzz test cases with '\' which will cause the vulnserver to crash but it needs to be there anyway, so I fuzzed with it.

```
s_initialize("VulnServer")
s_string("GMON \\", fuzzable=False)
s_string("FUZZ")

session.connect(s_get("VulnServer"))
session.fuzz()
```

Boofuzz Test Case 36 and 41 caused a crash. I didn't keep doing this as anything over 3950 bytes is going to cause a crash for this function.



GMON checks if the received buffer contains character '/' and if the length of the received buffer is larger than 3950 bytes, then calling Function3. So, we must send a buffer, which is at least 3951 bytes long to the Vulnserver, for buffer overflow to occur. Destination is 2008 bytes and if source is 3950+ it's going to cause an overflow when strcpy() moves the buffer to Destination.

```
else if ( !strncmp(buf, "GMON ", 5u) )
{
  *(_DWORD *)v10 = 1313819975;
  v11 = 1096045344;
  v12 = 1145394258;
  v13 = 10;
  for ( i = 5; i < len; ++i )
  {
    if ( buf[i] == '/' )
    {
      if ( strlen(buf) > 3950 )
        Function3(buf);
      break;
    }
  }
  v20 = send(s, v10, 13, 0);
}
```

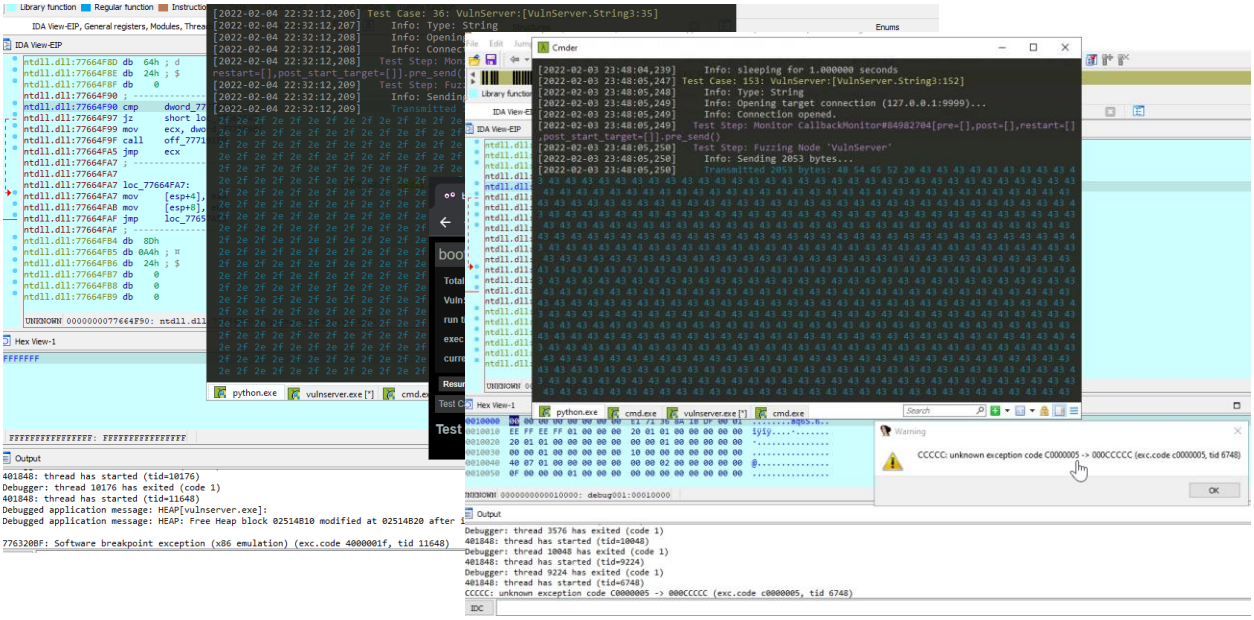
```
char *__cdecl Function3(char *Source)
{
  char Destination[2008]; // [esp+10h] [ebp-7D8h] BYREF

  return strcpy(Destination, Source);
}
```

Fuzzing The HTER Option

Boofuzz options

```
s_initialize("VulnServer")
s_string("HTER", fuzzable=False)
s_string("FUZZ")

session.connect(s_get("VulnServer"))
session.fuzz()
```

boofuzz Test Case 36, 41, 42, 153 caused a crash. I didn't keep doing this as anything over 1016 bytes is going to cause a crash for this function.



This one is straightforward. The call to vulnerable function, Function4, happens after the while loop and will happen with no conditions in the way if you make it inside this code block.

```
else if ( !strncmp(buf, "HTER ", 5u) )
{
  memset(String, 0, 3u);
  Destination = (char *)malloc(0x800u);
  memset(Destination, 0, 0x800u);
  i = 6;
  v18 = 0;
  while ( buf[i] && buf[i + 1] )
  {
    memcpy(String, &buf[i], 2u);
    v7 = strtoul(String, 0, 16);
    memset(&Destination[v18], v7, sizeof(char));
    i += 2;
    ++v18;
  }
  Function4(Destination);
  memset(Destination, 0, 0x800u);
  v20 = send(s, "HTER RUNNING FINE\n", 18, 0);
}
```

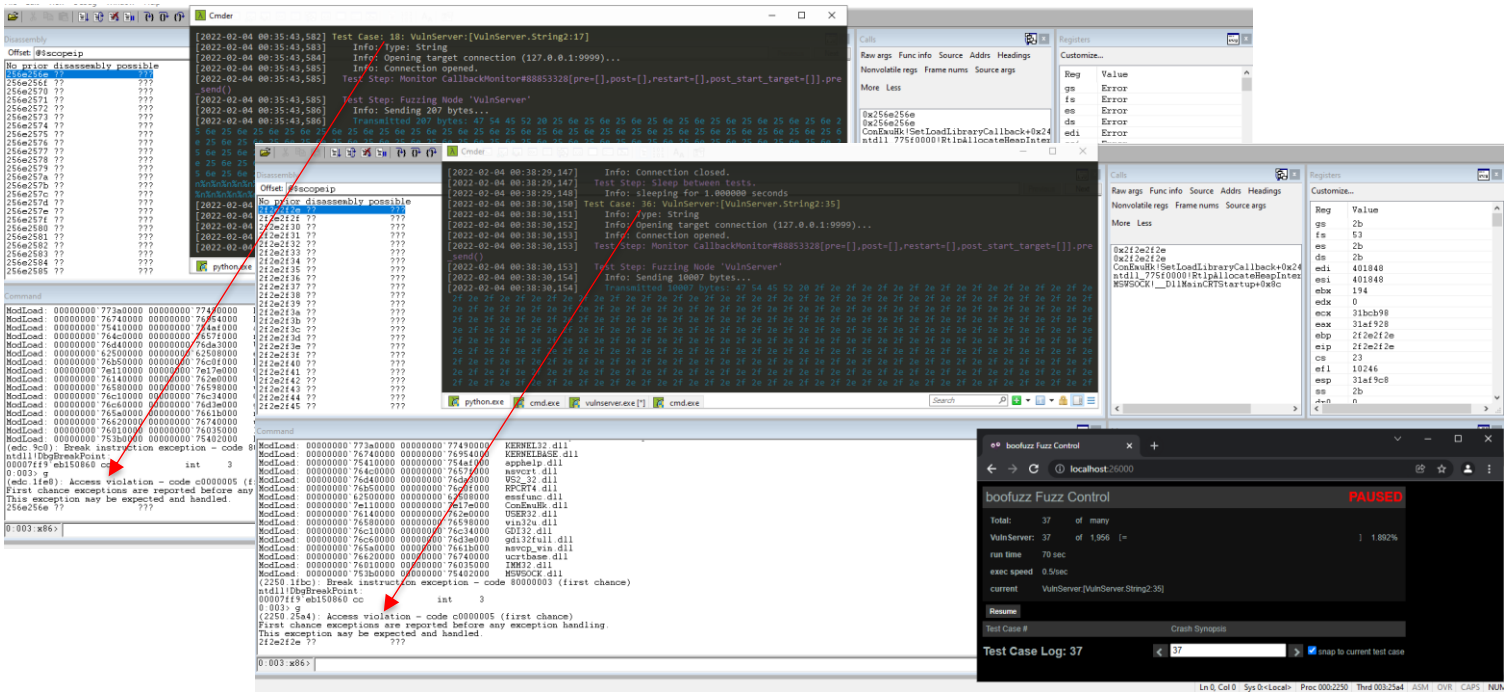
```
char *__cdecl Function4(char *Source)
{
  char Destination[1016]; // [esp+10h] [ebp-3F8h] BYREF

  return strcpy(Destination, Source);
}
```

Fuzzing The GTER Option

Boofuzz options

```
s_initialize("VulnServer")
s_string("GTER", fuzzable=False)
s_string("FUZZ")
s_static("\r\n")

session.connect(s_get("VulnServer"))
session.fuzz()
```

Boofuzz Test Case 18 and 36 caused a crash. I didn't keep doing this as anything over 152 bytes is going to cause a crash for this function.



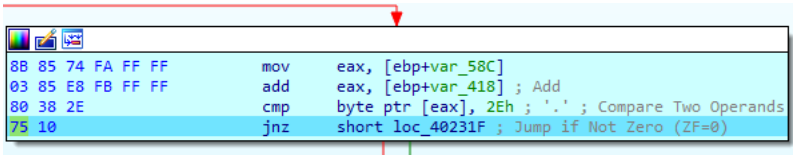
This one is straightforward. The call to vulnerable function, Function1, will crash with anything over 152 bytes sent to it.

```
else if ( !strncmp(buf, "GTER ", 5u) )
{
  Destination = (char *)malloc(0xB4u);
  memset(v22, 0, 0x400u);
  strncpy(Destination, buf, 0xB4u);
  memset(buf, 0, 0x1000u);
  Function1(Destination);
  v20 = send(s, "GTER ON TRACK\n", 14, 0);
}
```

```
char *__cdecl Function1(char *Source)
{
  char Destination[152]; // [esp+10h] [ebp-98h] BYREF

  return strcpy(Destination, Source);
}
```

Fuzzing The LTER Option

First you need to send “LTER .” to enter the proper vulnerable function. It checks that the first “argument” passed to the server following “LTER “ is a ‘.’ Then other text can follow.

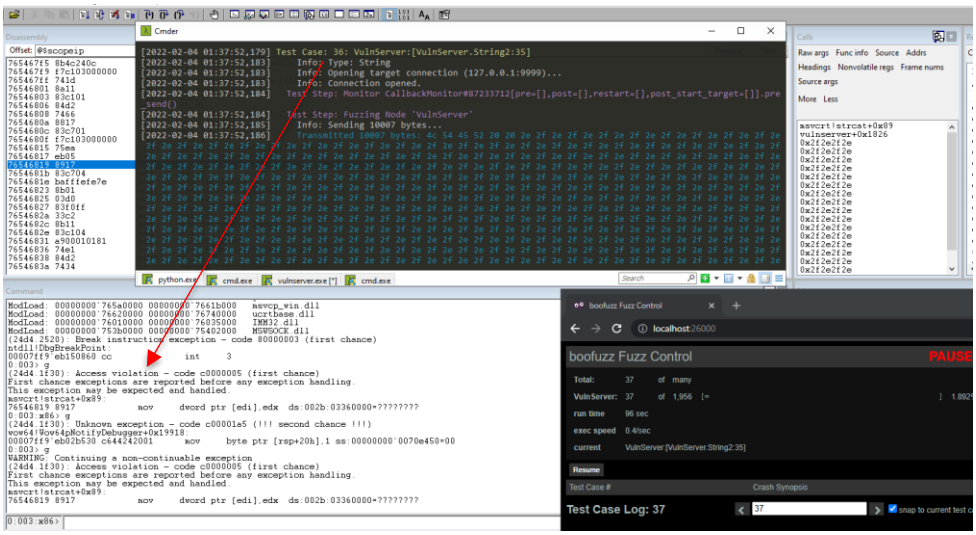


BooFuzz Options

```
s_initialize("VulnServer")
s_string("LTER .", fuzzable=False)
s_string("FUZZ")
s_static("\r\n")
```

Boofuzz Test Case 36 caused a crash. I didn’t keep doing this as anything over 2008 bytes is going to cause a crash for this function.

CSC 748 Software Exploitation: Lab 02



The call to vulnerable function, Function1 will crash with anything over 2008 bytes sent to it.

```
else if ( !strncmp(buf, "LTER ", 5u) )
{
  v8 = (char *)malloc(0x1000u);
  memset(v8, 0, 0x1000u);
  for ( i = 0; buf[i]; ++i )
  {
    if ( buf[i] >= 0 )
      v2 = buf[i];
    else
      v2 = buf[i] - 127;
    v8[i] = v2;
  }
  for ( i = 5; i <= 4095; ++i )
  {
    if ( v8[i] == 46 )
    {
      Function3(v8);
      break;
    }
  }
```

```
char *__cdecl Function3(char *Source)
{
  char Destination[2008]; // [esp+10h] [ebp-7D8h] BYREF

  return strcpy(Destination, Source);
}
```