Project2: Block It Up!

Nimoshika Jayaraman - UIN: 631000852

Sukanya Sravasti - UIN:431000993

We got an extension from the Professor until 11/25 without penalty. We used two target softwares for this project. Below are the links to the target softwares.

https://github.com/hummatli/onvif-qt-server-client

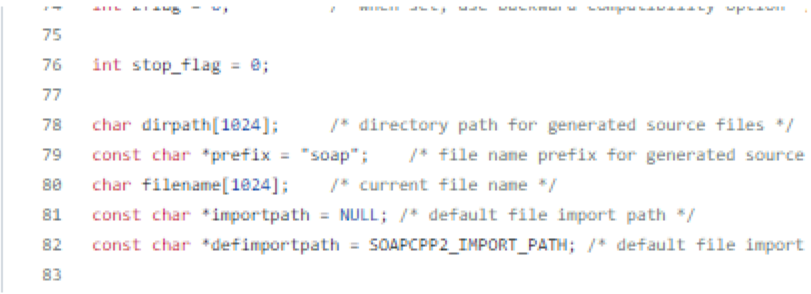
https://github.com/opencv

1) **Vulnerability Description:**

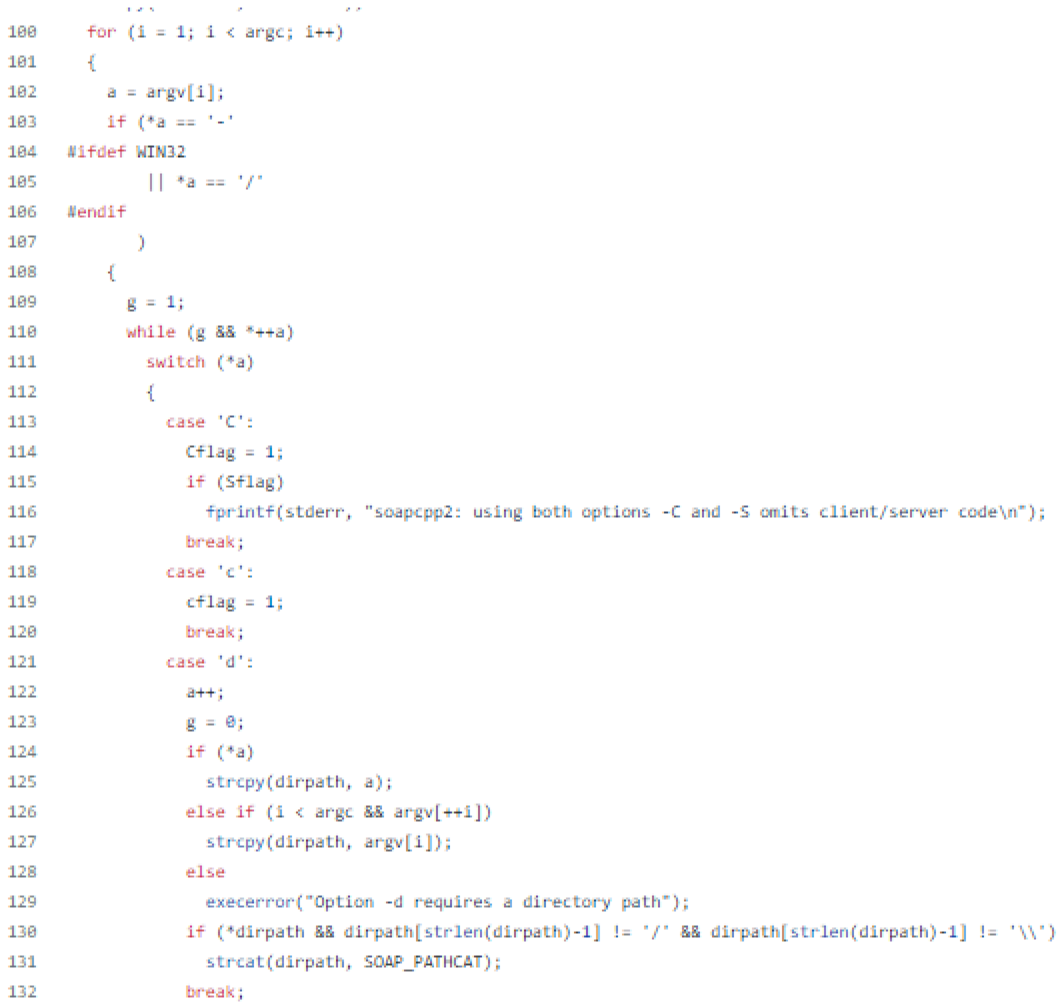
Buffer overflow happens when the volume of data from the source exceeds the storage capacity of the memory buffer. It overruns the buffer's boundary and overwrites the adjacent memory locations. This usually results in a stack smashing, affecting the availability to the users.

In location, **onvif-qt-server-client/onviflibs/gsoap/gsoap-2.8/gsoap/VisualStudio2005/soapcpp2/soapcpp2/soapcpp2.c -** line 125 buffer overflow happens when the string from "a" is trying to be copied to "dirpath" where the "dirpath" is limited by [1024] in line 78. When the input "a" has more than 1024, it causes the stack smashing. Screenshot of the code is attached below.

Category: Spatial Memory Attack(Buffer Overflow)



```
74   int zflag = 0;           /* when set, use backward compatibility option *
75
76   int stop_flag = 0;
77
78   char dirpath[1024];      /* directory path for generated source files */
79   const char *prefix = "soap";     /* file name prefix for generated source
80   char filename[1024];     /* current file name */
81   const char *importpath = NULL; /* default file import path */
82   const char *defimportpath = SOAPCPP2_IMPORT_PATH; /* default file import
83
```

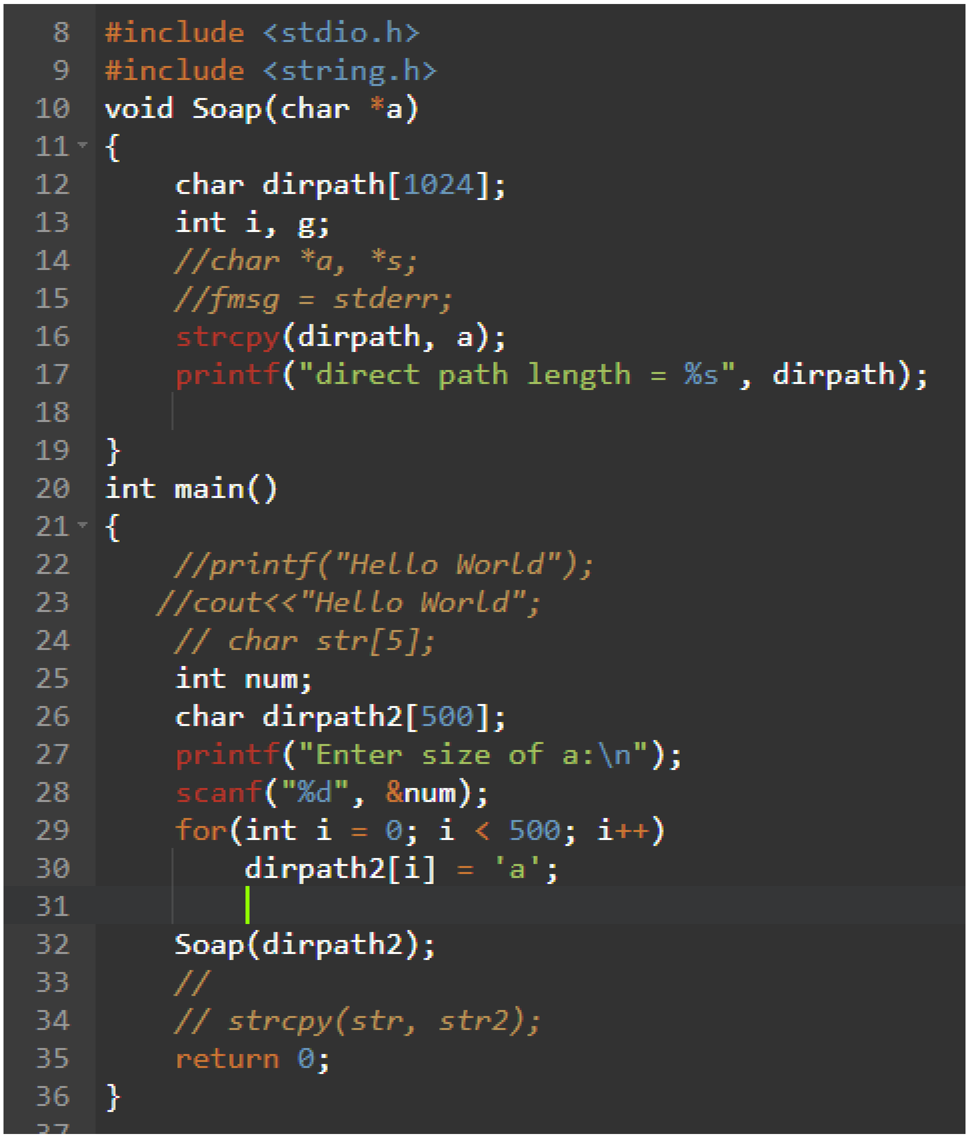


```
  for (i = 1; i < argc; i++)
  {
    a = argv[i];
    if (*a == '-'
#ifdef WIN32
        || *a == '/'
#endif
       )
    {
      g = 1;
      while (g && *++a)
        switch (*a)
        {
          case 'C':
            Cflag = 1;
            if (Sflag)
              fprintf(stderr, "soapcpp2: using both options -C and -S omits client/server code\n");
            break;
          case 'c':
            cflag = 1;
            break;
          case 'd':
            a++;
            g = 0;
            if (*a)
              strcpy(dirpath, a);
            else if (i < argc && argv[++i])
              strcpy(dirpath, argv[i]);
            else
              execerror("Option -d requires a directory path");
            if (*dirpath && dirpath[strlen(dirpath)-1] != '/' && dirpath[strlen(dirpath)-1] != '\\')
              strcat(dirpath, SOAP_PATHCAT);
            break;
```

Reproducing Process:

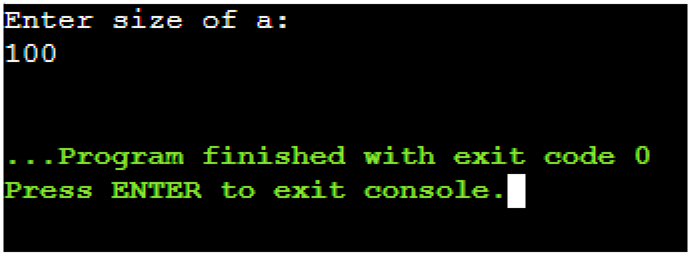
The important parts of the code responsible for the vulnerability are separated and reproduced in a separate function. Input is specified in such a way to cause buffer overflow and then stack smashing. Screen shot of the reproduced code and it's output while specifying an input within the bound and out of bound is demonstrated below. The corresponding output screenshots are attached. Online c++ code compiler was used for this demonstration.

Reproduced code:



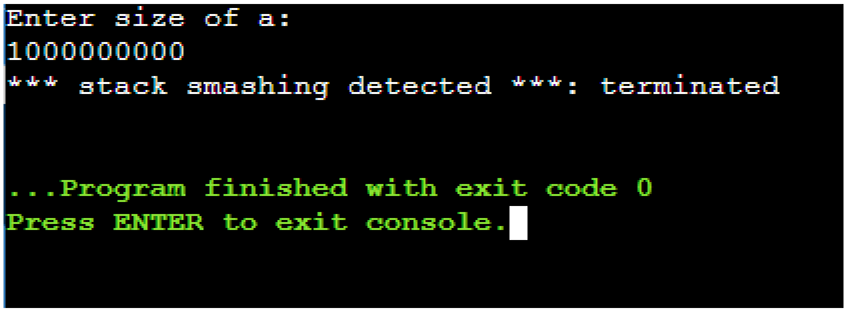
```c
#include <stdio.h>
#include <string.h>
void Soap(char *a)
{
    char dirpath[1024];
    int i, g;
    //char *a, *s;
    //fmsg = stderr;
    strcpy(dirpath, a);
    printf("direct path length = %s", dirpath);
    
}
int main()
{
    //printf("Hello World");
   //cout<<"Hello World";
    // char str[5];
    int num;
    char dirpath2[500];
    printf("Enter size of a:\n");
    scanf("%d", &num);
    for(int i = 0; i < 500; i++)
        dirpath2[i] = 'a';
        
    Soap(dirpath2);
    //
    // strcpy(str, str2);
    return 0;
}
```

Output when input is within the bound:



```
Enter size of a:
100


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when input was Out of Bound:



```
Enter size of a:
1000000000
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

Mitigation:
Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

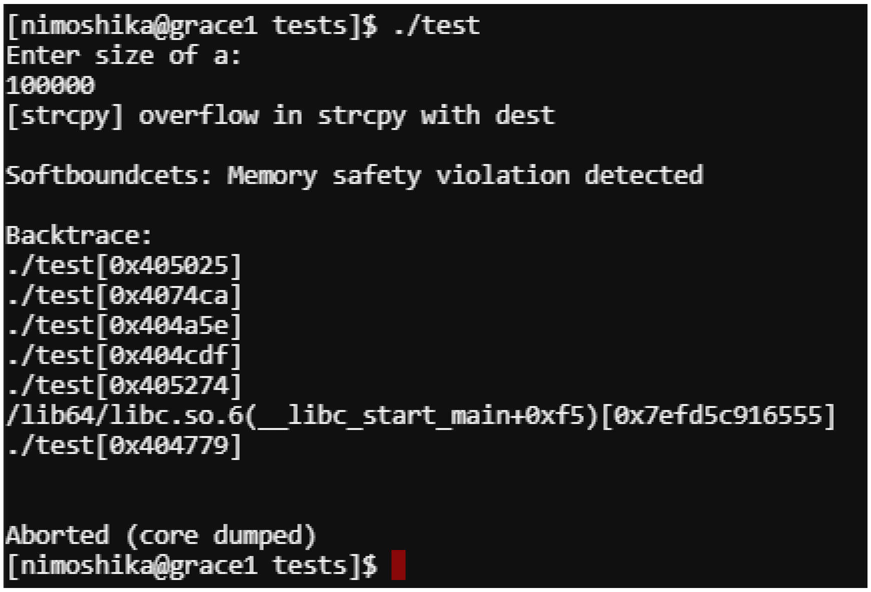
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

Demonstration Of Mitigation:

The following steps were followed for the demonstration:

1. `mkdir build; cd build`
2. `make -j8`
3. `export PATH=<git_repo>/llvm-38/build/bin:$PATH`
4. `make`
5. `clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt -lsoftboundcets_rt`
6. `./test`

Below is the screenshot of the mitigation when the input size is out of bound.



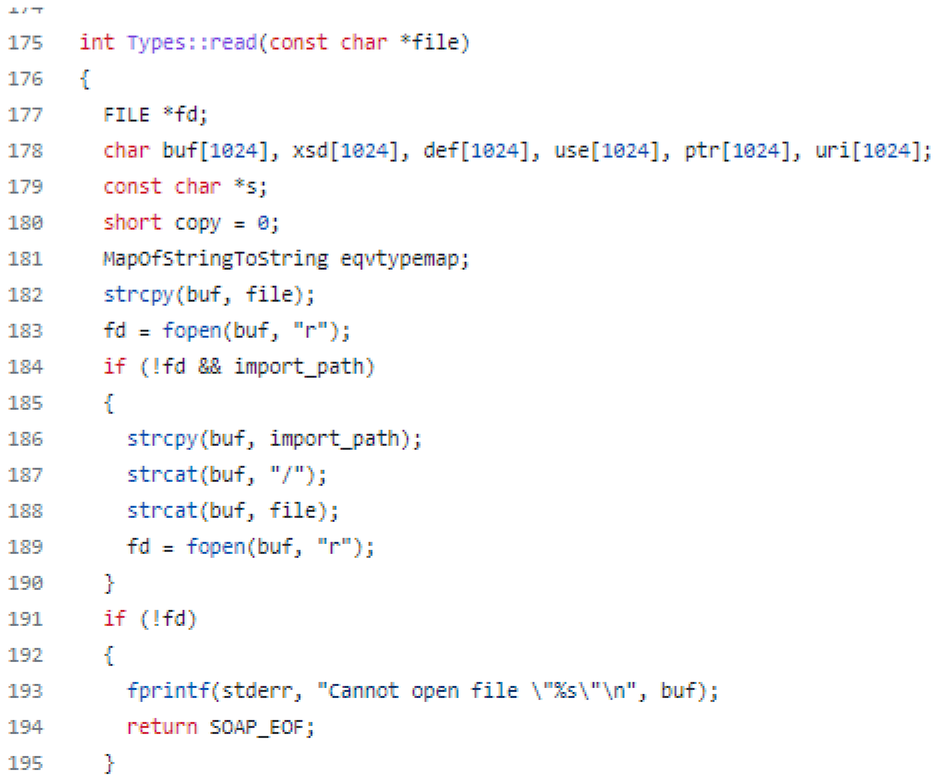
```
[nimoshika@grace1 tests]$ ./test
Enter size of a:
100000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405025]
./test[0x4074ca]
./test[0x404a5e]
./test[0x404cdf]
./test[0x405274]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7efd5c916555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$ 
```

2) **Vulnerability Description:**

Buffer overflow happens when the volume of data from the source exceeds the storage capacity of the memory buffer. It overruns the buffer's boundary and overwrites the adjacent memory locations. This usually results in a stack smashing, affecting the availability to the users.

In location, **onvif-qt-server-client/onviflibs/gsoap/gsoap-2.8/gsoap/VisualStudio2005/wsdl2h/wsdl2h/types.cpp -** line 182 buffer overflow happens when the string from "file" is trying to be copied to "buf" where the "buf" is limited by [1024] . When the input "file" has more than 1024, it causes the stack smashing. Screenshot of the code is attached below.

Category: Spatial Memory Attack(Buffer Overflow)

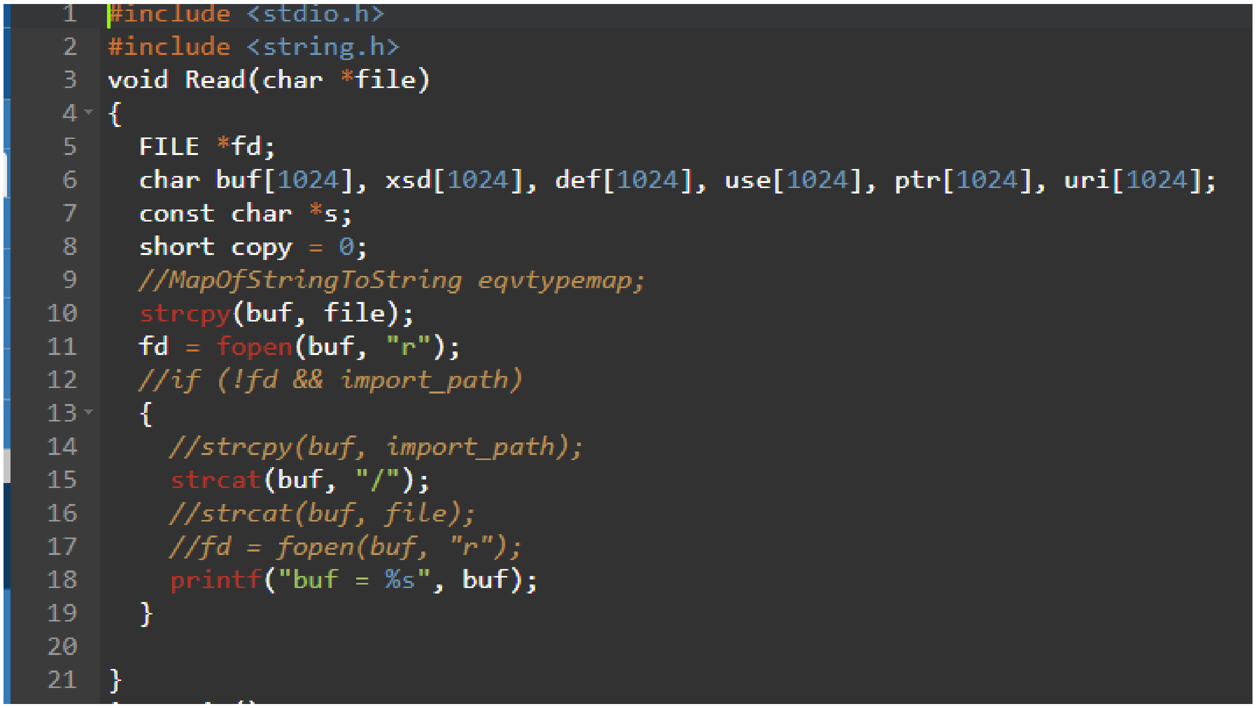


```
int Types::read(const char *file)
{
  FILE *fd;
  char buf[1024], xsd[1024], def[1024], use[1024], ptr[1024], uri[1024];
  const char *s;
  short copy = 0;
  MapOfStringToString eqvtypemap;
  strcpy(buf, file);
  fd = fopen(buf, "r");
  if (!fd && import_path)
  {
    strcpy(buf, import_path);
    strcat(buf, "/");
    strcat(buf, file);
    fd = fopen(buf, "r");
  }
  if (!fd)
  {
    fprintf(stderr, "Cannot open file \"%s\"\n", buf);
    return SOAP_EOF;
  }
```

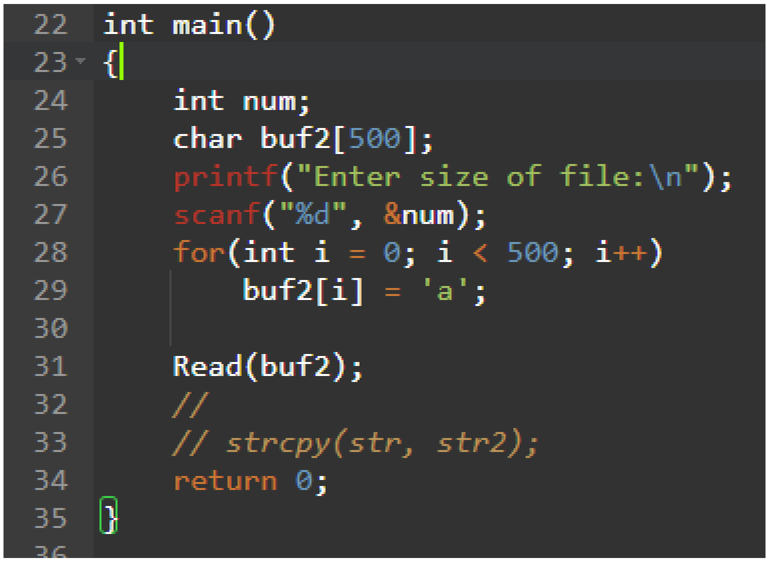
Reproducing Process:

The important parts of the code responsible for the vulnerability are separated and reproduced in a separate function. Input is specified in such a way to cause buffer overflow and then stack smashing. Screen shot of the reproduced code and it's output while specifying an input within the bound and out of bound is demonstrated below. The corresponding output screenshots are attached. Online c++ code compiler was used for this demonstration.

Reproduced Code:

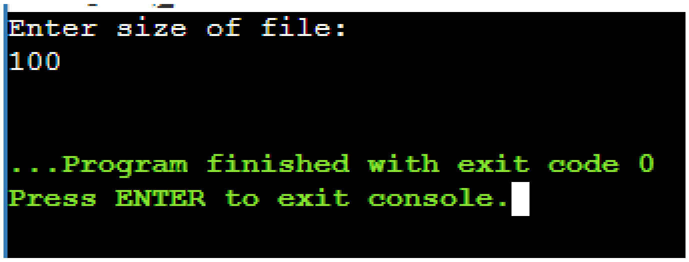


```c
#include <stdio.h>
#include <string.h>
void Read(char *file)
{
  FILE *fd;
  char buf[1024], xsd[1024], def[1024], use[1024], ptr[1024], uri[1024];
  const char *s;
  short copy = 0;
  //MapOfStringToString eqvtypemap;
  strcpy(buf, file);
  fd = fopen(buf, "r");
  //if (!fd && import_path)
  {
    //strcpy(buf, import_path);
    strcat(buf, "/");
    //strcat(buf, file);
    //fd = fopen(buf, "r");
    printf("buf = %s", buf);
  }

}
```



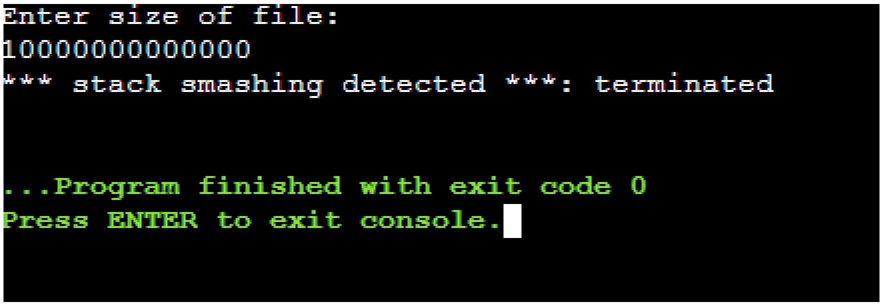
```c
int main()
{
    int num;
    char buf2[500];
    printf("Enter size of file:\n");
    scanf("%d", &num);
    for(int i = 0; i < 500; i++)
        buf2[i] = 'a';

    Read(buf2);
    //
    // strcpy(str, str2);
    return 0;
}
```

Output when the input is within the bound:



```
Enter size of file:
100


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when it causes stack smashing:



```
Enter size of file:
10000000000000
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

Mitigation:
Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

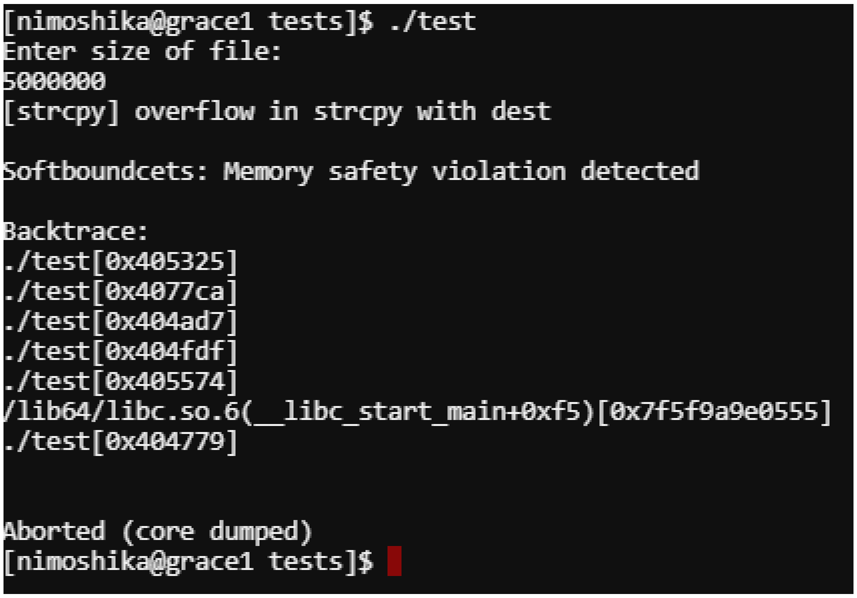
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

Demonstration Of Mitigation:

The following steps were followed for the demonstration:

1. `mkdir build; cd build`
2. `make -j8`
3. `export PATH=<git_repo>/llvm-38/build/bin:$PATH`
4. `make`
5. `clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt -lsoftboundcets_rt`
6. `./test`

Below is the screenshot of the mitigation when the input size is out of bound.



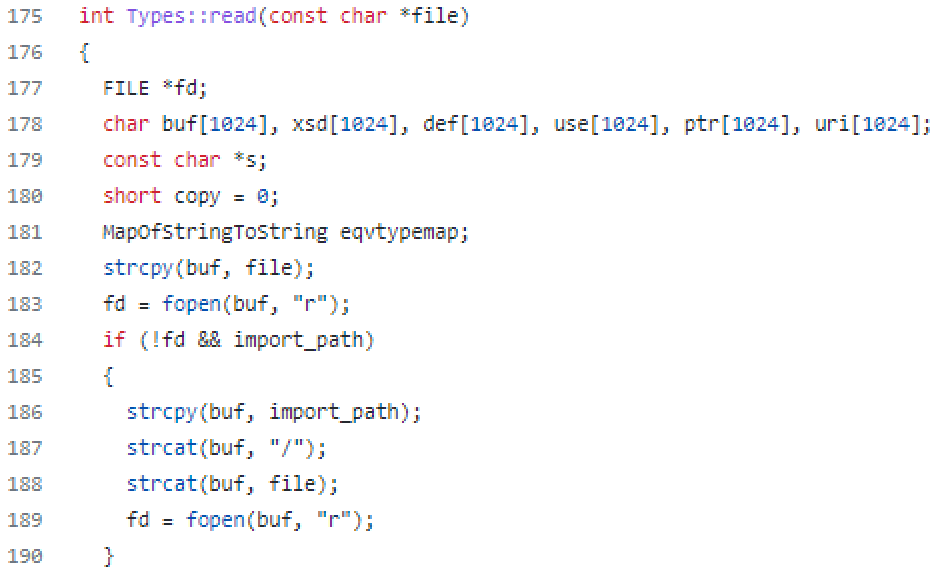
```
[nimoshika@grace1 tests]$ ./test
Enter size of file:
5000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405325]
./test[0x4077ca]
./test[0x404ad7]
./test[0x404fdf]
./test[0x405574]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7f5f9a9e0555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

3) **Vulnerability Description:**

Buffer overflow happens when the volume of data from the source exceeds the storage capacity of the memory buffer. It overruns the buffer's boundary and overwrites the adjacent memory locations. This usually results in a stack smashing, affecting the availability to the users.

In location, **onvif-qt-server-client/onviflibs/gsoap/gsoap-2.8/gsoap/VisualStudio2005/wsdl2h/wsdl2h/types.cpp-** line 186 buffer overflow happens when the string from "import_path" is trying to be copied to "buf" where the "import_path" is limited by [1024] . When the input "import_path" has more than 1024, it causes the stack smashing. Screenshot of the code is attached below.

Category: Spatial Memory Attack(Buffer Overflow)

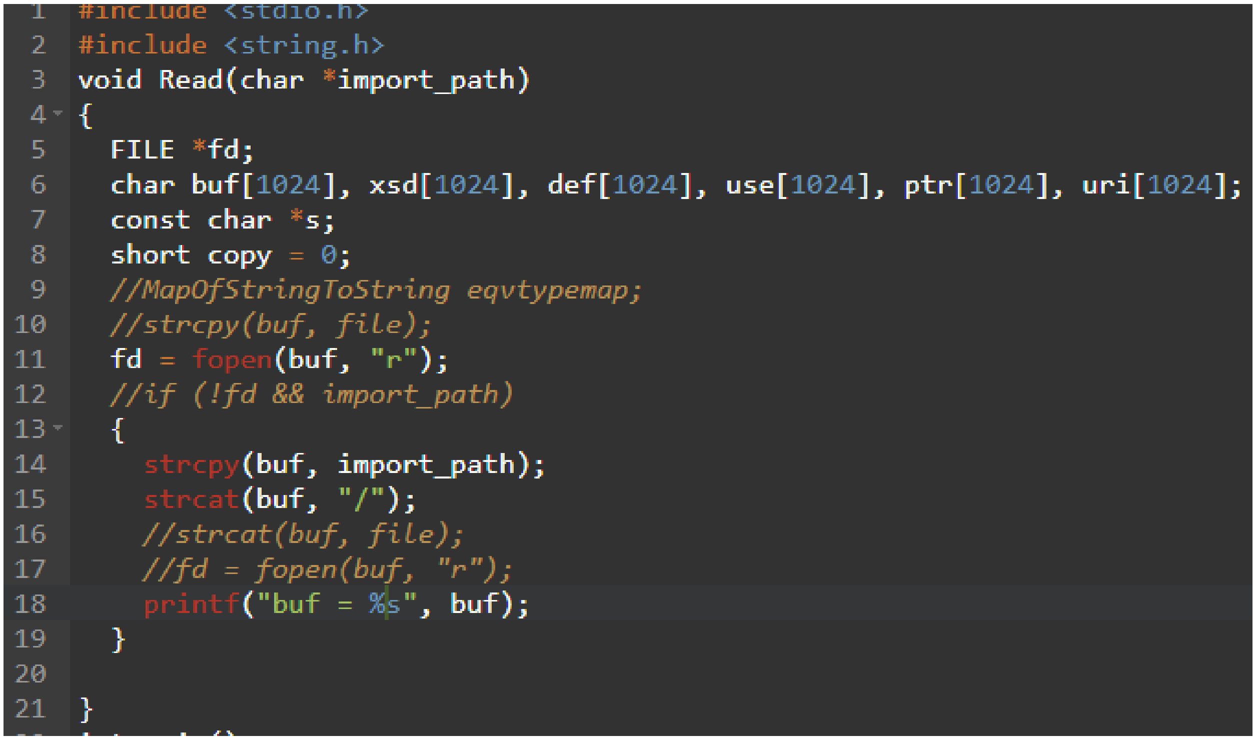


```
175   int Types::read(const char *file)
176   {
177     FILE *fd;
178     char buf[1024], xsd[1024], def[1024], use[1024], ptr[1024], uri[1024];
179     const char *s;
180     short copy = 0;
181     MapOfStringToString eqvtypemap;
182     strcpy(buf, file);
183     fd = fopen(buf, "r");
184     if (!fd && import_path)
185     {
186       strcpy(buf, import_path);
187       strcat(buf, "/");
188       strcat(buf, file);
189       fd = fopen(buf, "r");
190     }
```

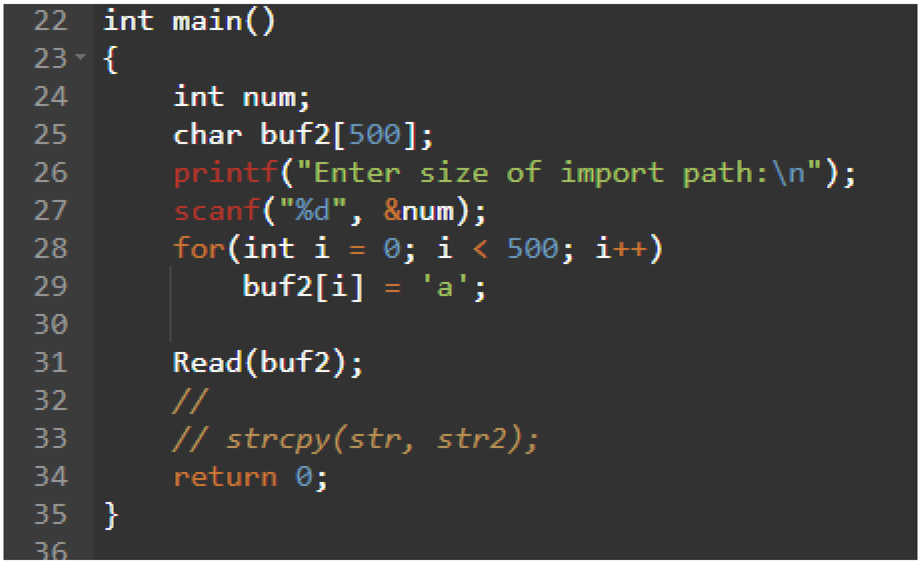
Reproducing Process:

The important parts of the code responsible for the vulnerability are separated and reproduced in a separate function. Input is specified in such a way to cause buffer overflow and then stack smashing. Screen shot of the reproduced code and it's output while specifying an input within the bound and out of bound is demonstrated below. The corresponding output screenshots are attached. Online c++ code compiler was used for this demonstration.

Reproduced code:

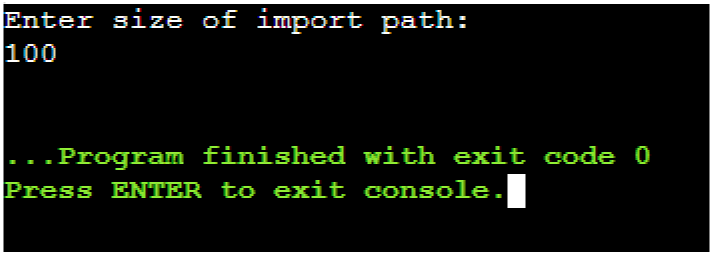


```
#include <stdio.h>
#include <string.h>
void Read(char *import_path)
{
  FILE *fd;
  char buf[1024], xsd[1024], def[1024], use[1024], ptr[1024], uri[1024];
  const char *s;
  short copy = 0;
  //MapOfStringToString eqvtypemap;
  //strcpy(buf, file);
  fd = fopen(buf, "r");
  //if (!fd && import_path)
  {
    strcpy(buf, import_path);
    strcat(buf, "/");
    //strcat(buf, file);
    //fd = fopen(buf, "r");
    printf("buf = %s", buf);
  }

}
```



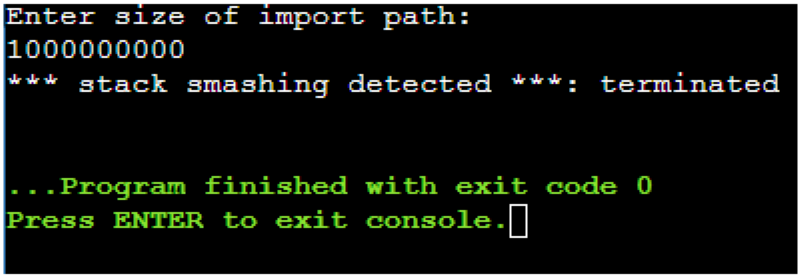
```
int main()
{
    int num;
    char buf2[500];
    printf("Enter size of import path:\n");
    scanf("%d", &num);
    for(int i = 0; i < 500; i++)
        buf2[i] = 'a';

    Read(buf2);
    //
    // strcpy(str, str2);
    return 0;
}
```

Output When input is within the bound:



```
Enter size of import path:
100


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when it causes stack smashing:



```
Enter size of import path:
1000000000
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

Mitigation:

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

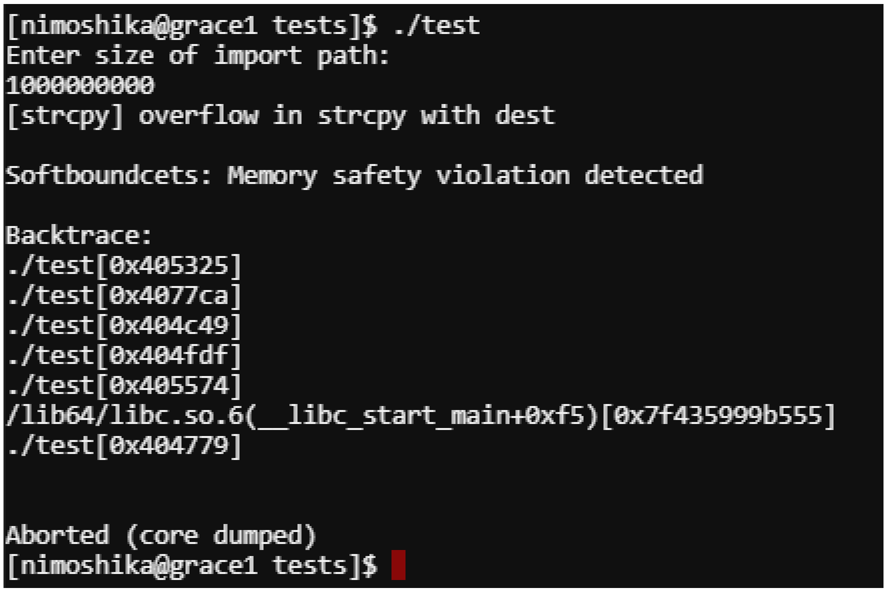
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

Demonstration Of Mitigation:

The following steps were followed for the demonstration:

```
1. mkdir build; cd build
2. make -j8
3. export PATH=<git_repo>/llvm-38/build/bin:$PATH
4. make
5. clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
6. ./test
```

Below is the screenshot of the mitigation when the input size is out of bound.



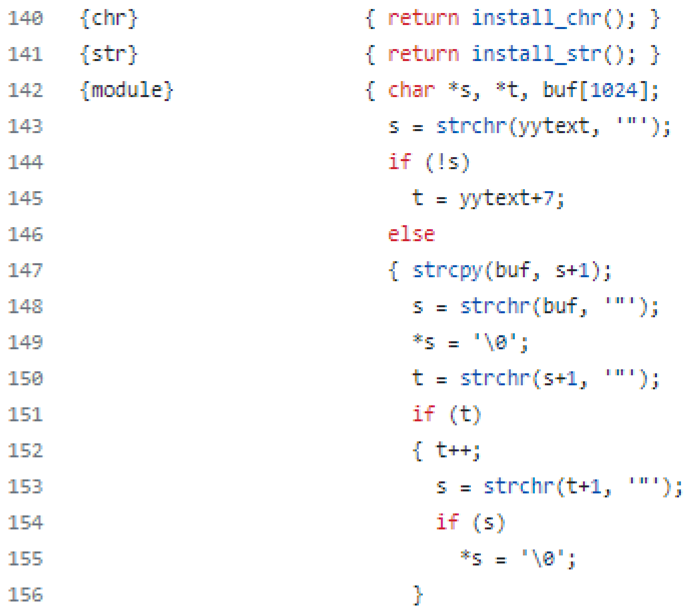
```
[nimoshika@grace1 tests]$ ./test
Enter size of import path:
1000000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405325]
./test[0x4077ca]
./test[0x404c49]
./test[0x404fdf]
./test[0x405574]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7f435999b555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

4) **Vulnerability Description:**

Buffer overflow happens when the volume of data from the source exceeds the storage capacity of the memory buffer. It overruns the buffer's boundary and overwrites the adjacent memory locations. This usually results in a stack smashing, affecting the availability to the users.

In location, **onvif-qt-server-client/onviflibs/gsoap/gsoap-2.8/gsoap/src/**soapcpp2_lex.l
- line 147 buffer overflow happens when the string from “import_path” is trying to be copied to “buf” where the “import_path” is limited by [1024] . When the input “import_path” has more than 1024, it causes the stack smashing. Screenshot of the code is attached below.

<u>Category:</u> Spatial Memory Attack(Buffer Overflow)

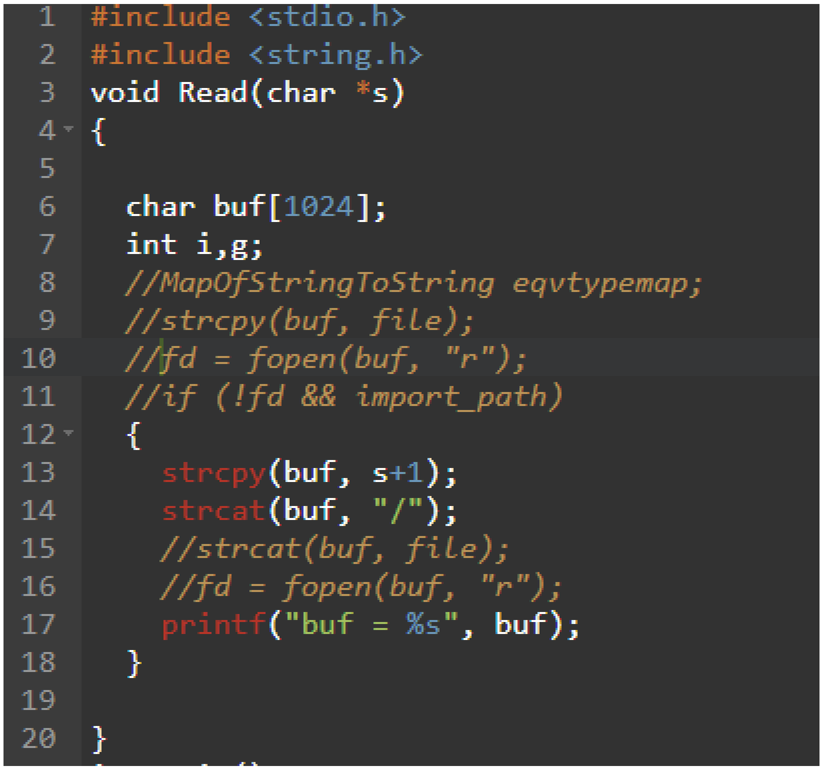


```
140  {chr}                    { return install_chr(); }
141  {str}                    { return install_str(); }
142  {module}                 { char *s, *t, buf[1024];
143                             s = strchr(yytext, '"');
144                             if (!s)
145                               t = yytext+7;
146                             else
147                             { strcpy(buf, s+1);
148                               s = strchr(buf, '"');
149                               *s = '\0';
150                               t = strchr(s+1, '"');
151                               if (t)
152                               { t++;
153                                 s = strchr(t+1, '"');
154                                 if (s)
155                                   *s = '\0';
156                               }
```

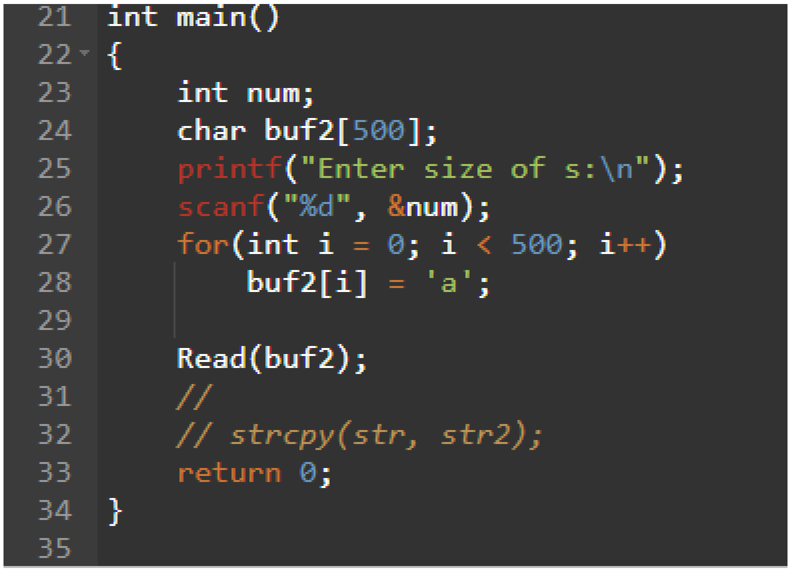
<u>Reproducing Process:</u>

The important parts of the code responsible for the vulnerability are separated and reproduced in a separate function. Input is specified in such a way to cause buffer overflow and then stack smashing. Screen shot of the reproduced code and it’s output while specifying an input within the bound and out of bound is demonstrated below. The corresponding output screenshots are attached. Online c++ code compiler was used for this demonstration.

<u>Reproduced code:</u>

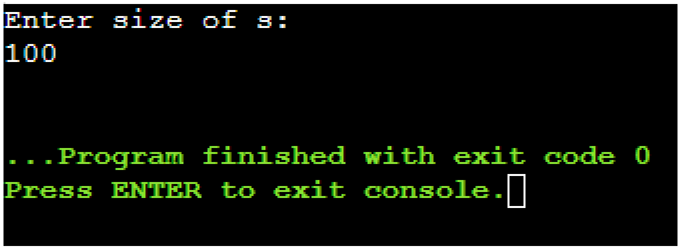


```c
#include <stdio.h>
#include <string.h>
void Read(char *s)
{

  char buf[1024];
  int i,g;
  //MapOfStringToString eqvtypemap;
  //strcpy(buf, file);
  //fd = fopen(buf, "r");
  //if (!fd && import_path)
  {
    strcpy(buf, s+1);
    strcat(buf, "/");
    //strcat(buf, file);
    //fd = fopen(buf, "r");
    printf("buf = %s", buf);
  }

}
```



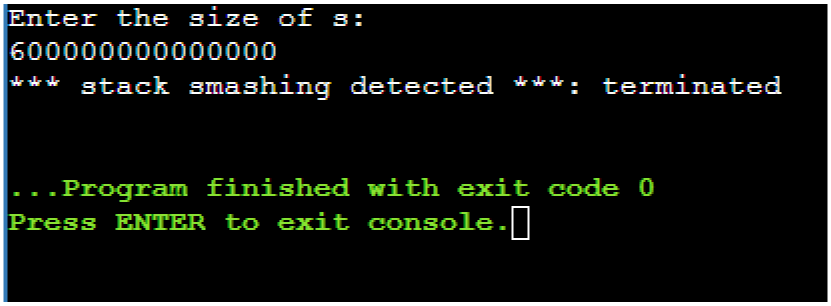
```c
int main()
{
    int num;
    char buf2[500];
    printf("Enter size of s:\n");
    scanf("%d", &num);
    for(int i = 0; i < 500; i++)
        buf2[i] = 'a';

    Read(buf2);
    //
    // strcpy(str, str2);
    return 0;
}

```

Output when the input is within the bound:



```
Enter size of s:
100


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when the stack smashing happens:



```
Enter the size of s:
600000000000000
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

Mitigation:

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

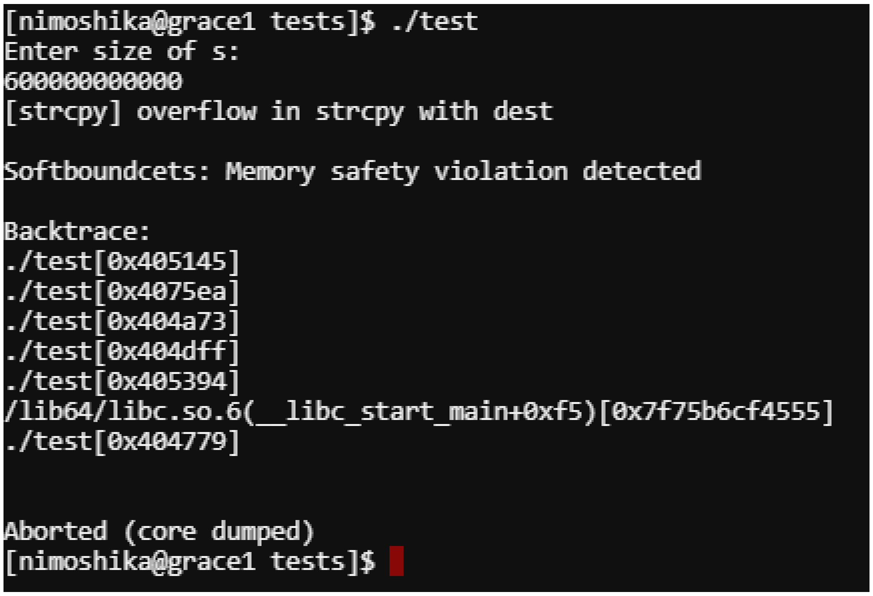
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

Demonstration Of Mitigation:

The following steps were followed for the demonstration:

```
7. mkdir build; cd build
8. make -j8
9. export PATH=<git_repo>/llvm-38/build/bin:$PATH
10.make
11.clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
12../test
```

Below is the screenshot of the mitigation when the input size is out of bound. Below is the screenshot of the mitigation when the input size is out of bound. The error was discovered and it was aborted, which protects the program.



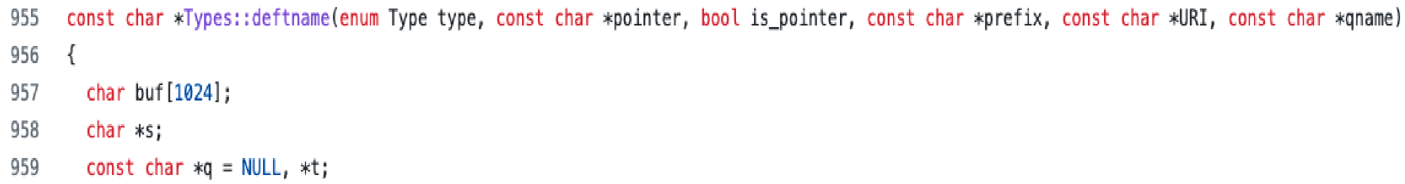
```
[nimoshika@grace1 tests]$ ./test
Enter size of s:
600000000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405145]
./test[0x4075ea]
./test[0x404a73]
./test[0x404dff]
./test[0x405394]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7f75b6cf4555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

5)

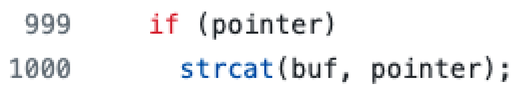
- **Vulnerability Description:**

A buffer overflow happens when a program tries to write more data to a fixed length buffer beyond the buffer's capacity. This overwrites data in adjacent memory locations. This in turn can corrupt data in that adjacent space. This can corrupt data and cause system crashes. This also creates an opportunity for the attacker to execute arbitrary code, which is a vulnerability.

The vulnerability was detected through flawfinder. In the Github directory file onvif-qt-server-client/onviflibs/gsoap/gsoap-2.8/gsoap/VisualStudio2005/wsdl2h/wsdl2h/types.cpp (screenshot below), in line 1000 the source array "pointer" is copied to "buf" (the destination array of fixed size 1024). A buffer overflow occurs when the size of "pointer" is greater than the size of "buf."

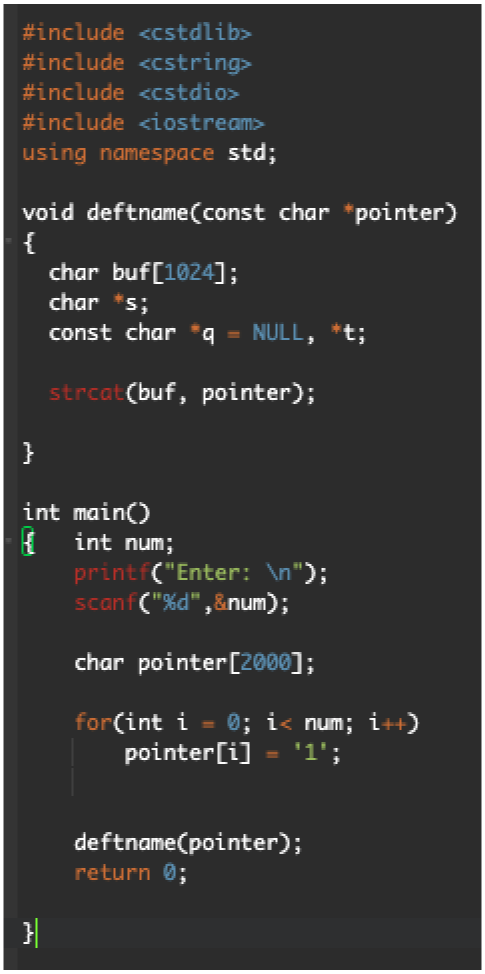


```
955  const char *Types::deftname(enum Type type, const char *pointer, bool is_pointer, const char *prefix, const char *URI, const char *qname)
956  {
957    char buf[1024];
958    char *s;
959    const char *q = NULL, *t;
```



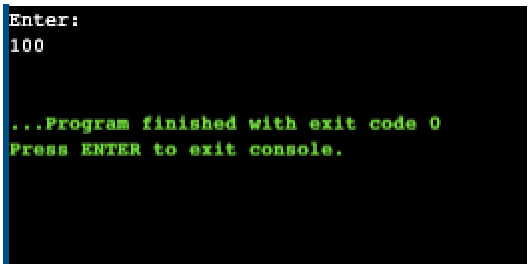
```
999      if (pointer)
1000       strcat(buf, pointer);
```

- **Category of attack:** Spatial memory (buffer overflow)
- **Reproducing process:** For the reproduction of the vulnerability, we separated the function which caused the vulnerability and compiled it separately. In doing so, we kept only the variables which were relevant to the vulnerability to avoid compiling errors. We then passed inputs to the reproduced function to expose the vulnerability.
  As shown below, the vulnerability is discovered once an input greater than the size of the destination buffer is passed to the function.



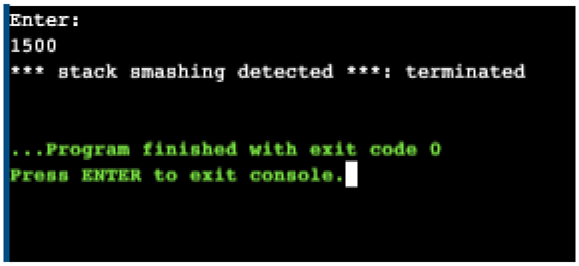
```
#include <cstdlib>
#include <cstring>
#include <cstdio>
#include <iostream>
using namespace std;

void deftname(const char *pointer)
{
  char buf[1024];
  char *s;
  const char *q = NULL, *t;

  strcat(buf, pointer);

}

int main()
{   int num;
    printf("Enter: \n");
    scanf("%d",&num);

    char pointer[2000];

    for(int i = 0; i< num; i++)
        pointer[i] = '1';


    deftname(pointer);
    return 0;

}
```

Output when the input is within the bound:



```
Enter:
100


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when the stack smashing happens:



```
Enter:
1500
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

- **Mitigation:**

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

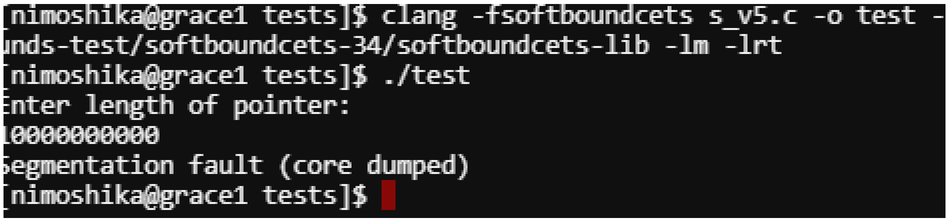
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

- **Demonstration Of Mitigation:**

The following steps were followed for the demonstration:

```
13.mkdir build; cd build
14.make -j8
15.export PATH=<git_repo>/llvm-38/build/bin:$PATH
16.make
17.clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
18../test
```

Below is the screenshot of the mitigation when the input size is out of bound. The error was discovered and it was aborted, which protects the program.

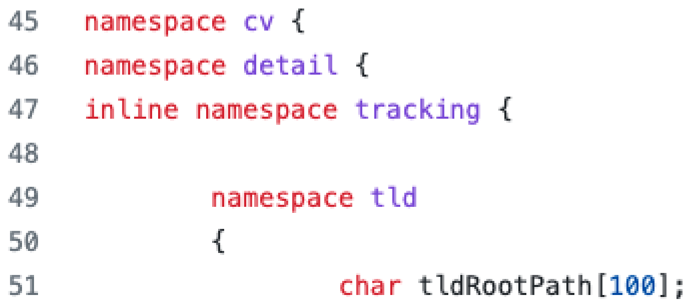


```
[nimoshika@grace1 tests]$ clang -fsoftboundcets s_v5.c -o test -
unds-test/softboundcets-34/softboundcets-lib -lm -lrt
[nimoshika@grace1 tests]$ ./test
Enter length of pointer:
10000000000
Segmentation fault (core dumped)
[nimoshika@grace1 tests]$
```

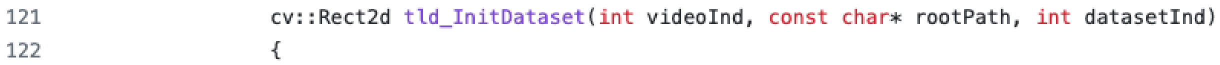
6)

- **Vulnerability Description:** A buffer overflow happens when a program tries to write more data to a fixed length buffer beyond the buffer's capacity. This overwrites data in adjacent memory locations. This in turn can corrupt data in that adjacent space. This can corrupt data and cause system crashes. This also creates an opportunity for the attacker to execute arbitrary code, which is a vulnerability.

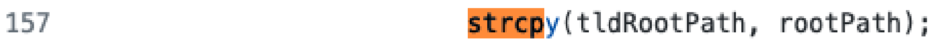
The vulnerability was detected through flawfinder. In the Github directory file opencv_contrib/modules/tracking/src/tldDataset.cpp (screenshot below), in line 157 the source array "rootPath" is copied to "tldRootPath" (the destination array of fixed size 100). A buffer overflow occurs when the size of "rootPath" is greater than the size of "tldRootPath."



```
45   namespace cv {
46   namespace detail {
47   inline namespace tracking {
48
49           namespace tld
50           {
51                   char tldRootPath[100];
```

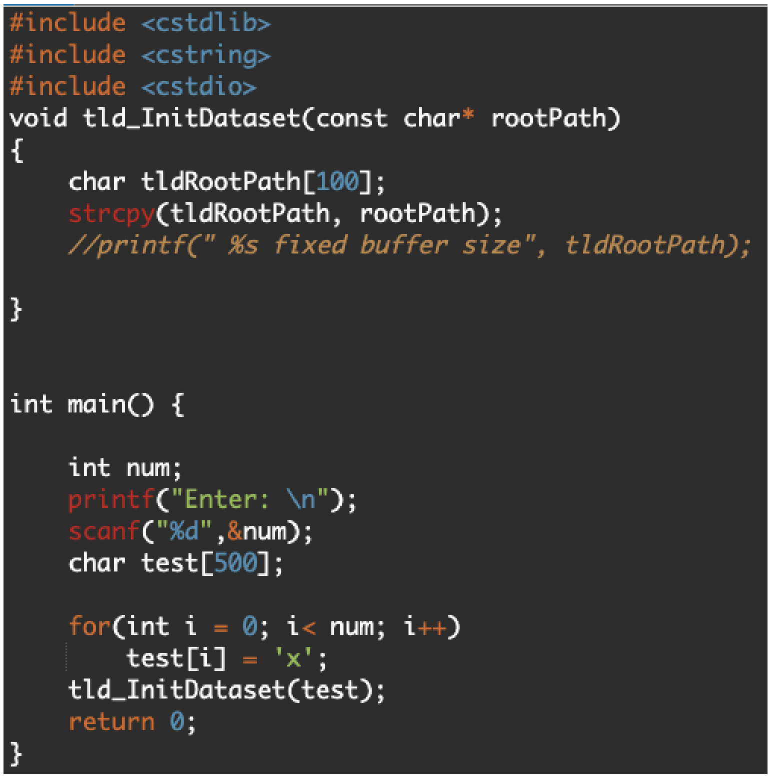


```
121                  cv::Rect2d tld_InitDataset(int videoInd, const char* rootPath, int datasetInd)
122                  {
```



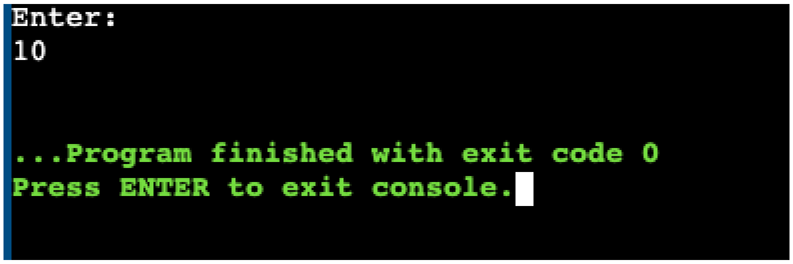
```
157                          strcpy(tldRootPath, rootPath);
```

- **Category of attack:** Spatial memory (buffer overflow)
- **Reproducing process:** For the reproduction of the vulnerability, we separated the function which caused the vulnerability and compiled it separately. In doing so, we kept only the variables which were relevant to the vulnerability to avoid compiling errors. We then passed inputs to the reproduced function to expose the vulnerability.
  As shown below, the vulnerability is discovered once an input greater than the size of the destination buffer is passed to the function.



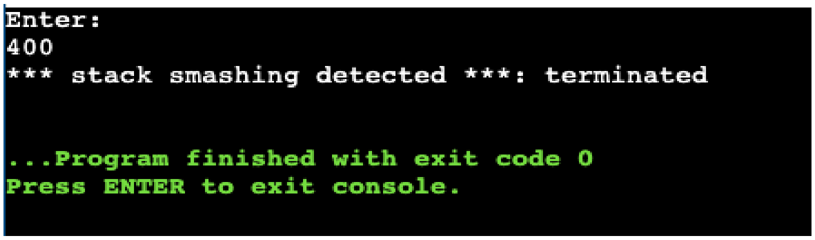
```
#include <cstdlib>
#include <cstring>
#include <cstdio>
void tld_InitDataset(const char* rootPath)
{
    char tldRootPath[100];
    strcpy(tldRootPath, rootPath);
    //printf(" %s fixed buffer size", tldRootPath);

}


int main() {

    int num;
    printf("Enter: \n");
    scanf("%d",&num);
    char test[500];

    for(int i = 0; i< num; i++)
        test[i] = 'x';
    tld_InitDataset(test);
    return 0;
}
```

Output when the input is within the bound:



```
Enter:
10


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when the stack smashing happens:



```
Enter:
400
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

- **Mitigation:**

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

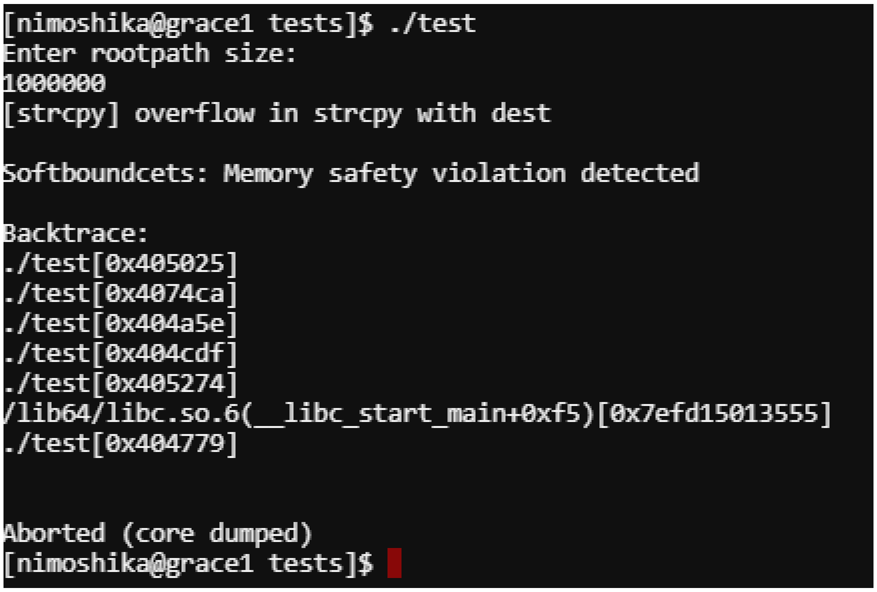
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

- **Demonstration Of Mitigation:**

The following steps were followed for the demonstration:

```
19.mkdir build; cd build
20.make -j8
21.export PATH=<git_repo>/llvm-38/build/bin:$PATH
22.make
23.clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
24../test
```

Below is the screenshot of the mitigation when the input size is out of bound. The error was discovered and it was aborted, which protects the program.



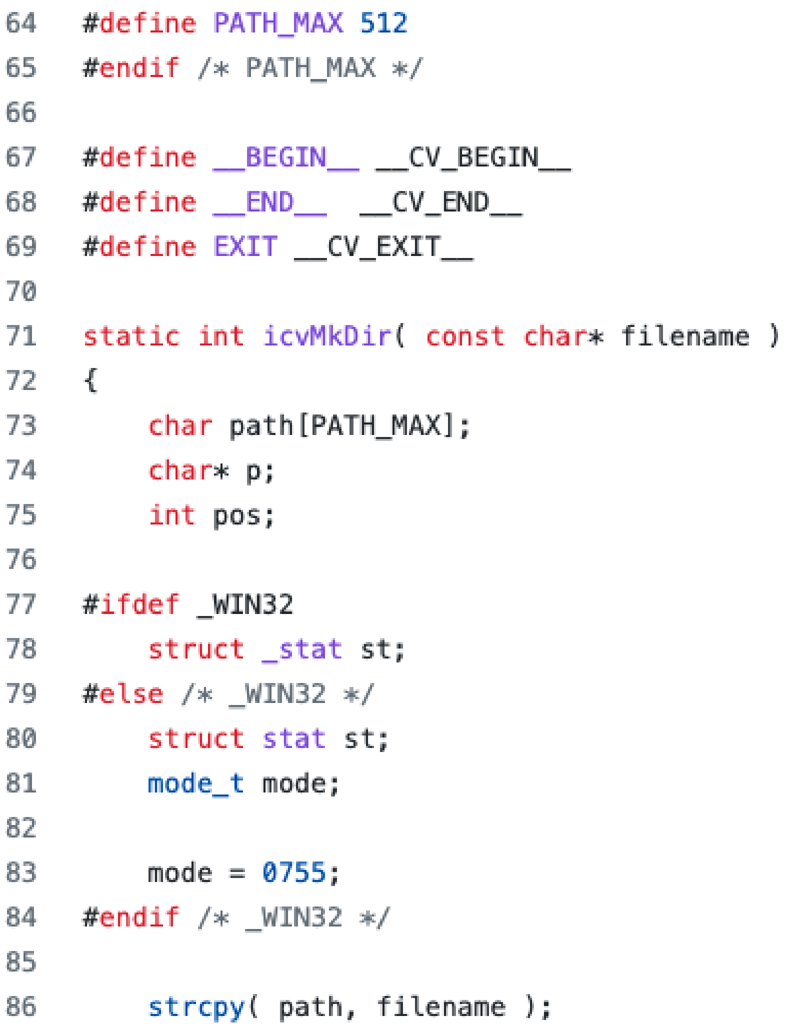
```
[nimoshika@grace1 tests]$ ./test
Enter rootpath size:
1000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405025]
./test[0x4074ca]
./test[0x404a5e]
./test[0x404cdf]
./test[0x405274]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7efd15013555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

7)

- **Vulnerability Description:**

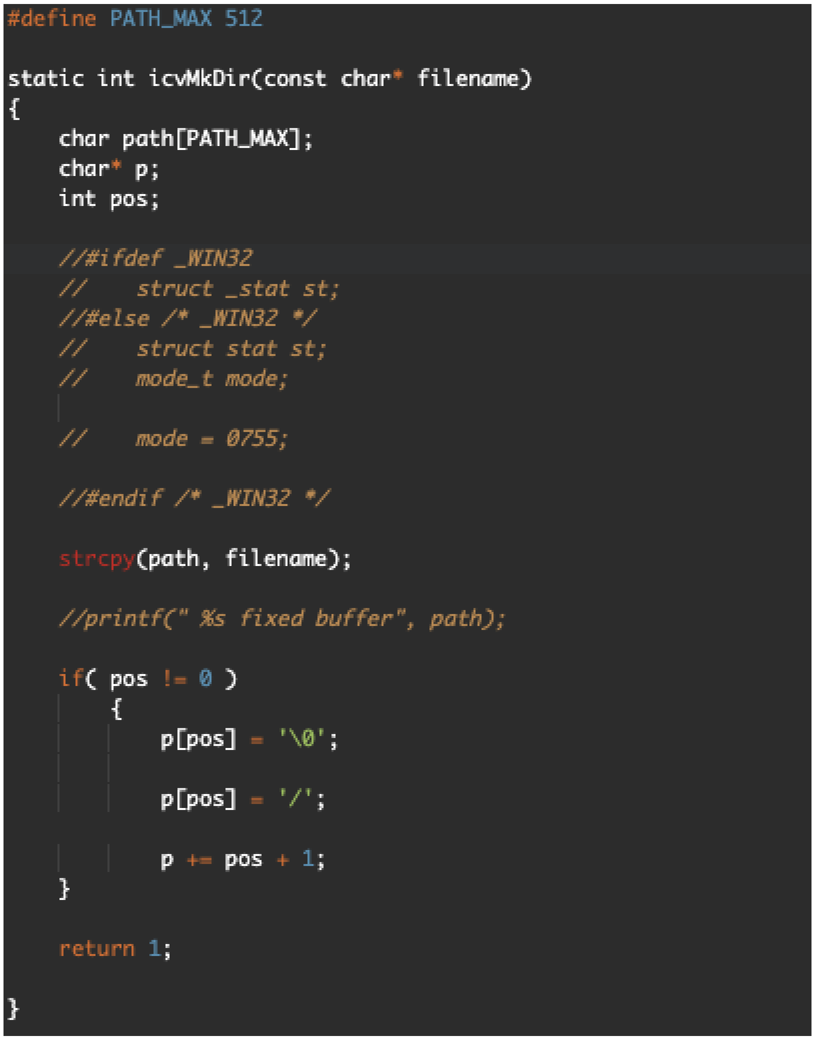
A buffer overflow happens when a program tries to write more data to a fixed length buffer beyond the buffer’s capacity. This overwrites data in adjacent memory locations. This in turn can corrupt data in that adjacent space. This can corrupt data and cause system crashes. This also creates an opportunity for the attacker to execute arbitrary code, which is a vulnerability.

The vulnerability was detected through flawfinder. In the Github directory file opencv/apps/createsamples/utility.cpp (screenshot below), in line 86 the source array “filename” is copied to “path” (the destination array of fixed size 512). A buffer overflow occurs when the size of “filename” is greater than the size of “path.”

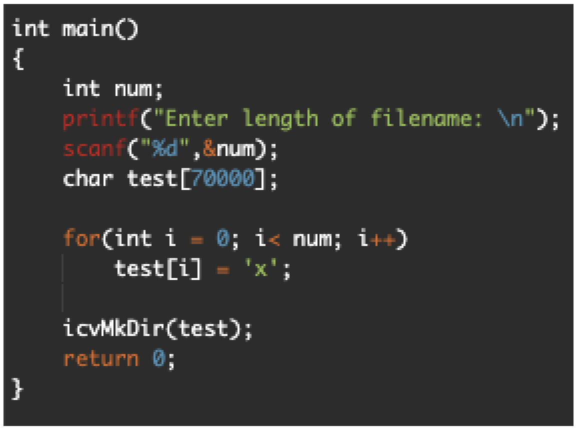


```
#define PATH_MAX 512
#endif /* PATH_MAX */

#define __BEGIN__ __CV_BEGIN__
#define __END__  __CV_END__
#define EXIT __CV_EXIT__

static int icvMkDir( const char* filename )
{
    char path[PATH_MAX];
    char* p;
    int pos;

#ifdef _WIN32
    struct _stat st;
#else /* _WIN32 */
    struct stat st;
    mode_t mode;

    mode = 0755;
#endif /* _WIN32 */

    strcpy( path, filename );
```

- **Category of attack:** Spatial memory (buffer overflow)
- **Reproducing process:** For the reproduction of the vulnerability, we separated the function which caused the vulnerability and compiled it separately. In doing so, we kept only the variables which were relevant to the vulnerability to avoid compiling errors. We then passed inputs to the reproduced function to expose the vulnerability.
  As shown below, the vulnerability is discovered once an input greater than the size of the destination buffer is passed to the function.

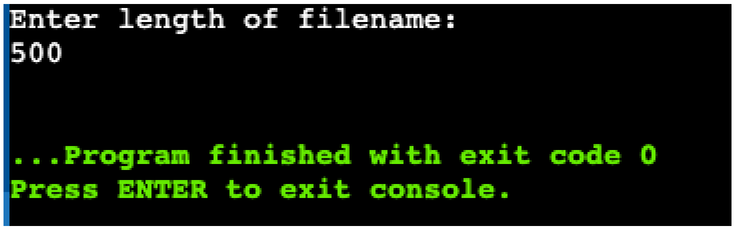


```
#define PATH_MAX 512

static int icvMkDir(const char* filename)
{
    char path[PATH_MAX];
    char* p;
    int pos;

    //#ifdef _WIN32
    //    struct _stat st;
    //#else /* _WIN32 */
    //    struct stat st;
    //    mode_t mode;

    //    mode = 0755;

    //#endif /* _WIN32 */

    strcpy(path, filename);

    //printf(" %s fixed buffer", path);

    if( pos != 0 )
        {
            p[pos] = '\0';

            p[pos] = '/';

            p += pos + 1;
    }

    return 1;

}
```



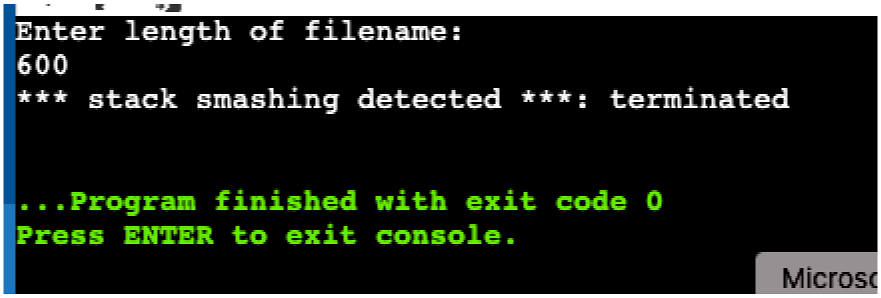
```
int main()
{
    int num;
    printf("Enter length of filename: \n");
    scanf("%d",&num);
    char test[70000];

    for(int i = 0; i< num; i++)
        test[i] = 'x';

    icvMkDir(test);
    return 0;
}
```

Output when the input is within the bound:



```
Enter length of filename:
500


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when the stack smashing happens:



```
Enter length of filename:
600
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

- **Mitigation:**

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

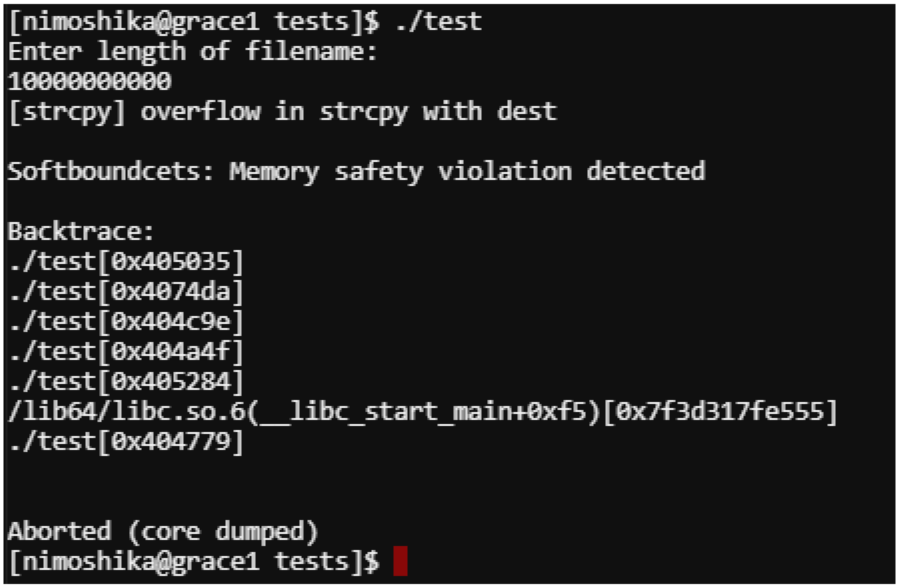
- **Demonstration Of Mitigation:**

The following steps were followed for the demonstration:

```
25.mkdir build; cd build
26.make -j8
27.export PATH=<git_repo>/llvm-38/build/bin:$PATH
28.make
29.clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
```

```
30../test
```

Below is the screenshot of the mitigation when the input size is out of bound. The error was discovered and it was aborted, which protects the program.



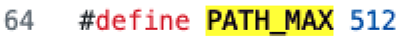
```
[nimoshika@grace1 tests]$ ./test
Enter length of filename:
10000000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405035]
./test[0x4074da]
./test[0x404c9e]
./test[0x404a4f]
./test[0x405284]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7f3d317fe555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

8)

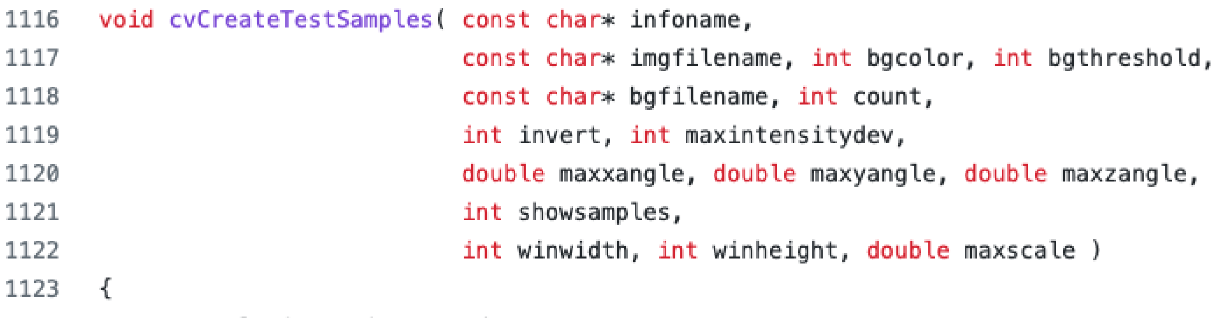
- **Vulnerability Description:**

A buffer overflow happens when a program tries to write more data to a fixed length buffer beyond the buffer's capacity. This overwrites data in adjacent memory locations. This in turn can corrupt data in that adjacent space. This can corrupt data and cause system crashes. This also creates an opportunity for the attacker to execute arbitrary code, which is a vulnerability.

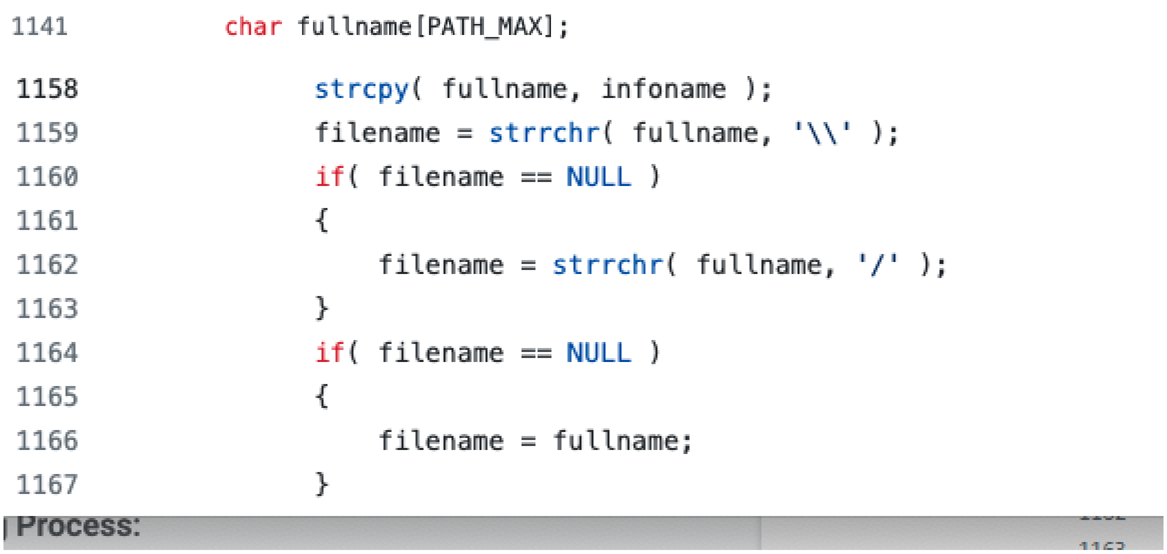
The vulnerability was detected through flawfinder. In the Github directory file opencv/apps/createsamples/utility.cpp (screenshot below), in line 1158 the source array "infoname" is copied to "fullname" (the destination array of fixed size 512). A buffer overflow occurs when the size of "filename" is greater than the size of "path."



```
64   #define PATH_MAX 512
```

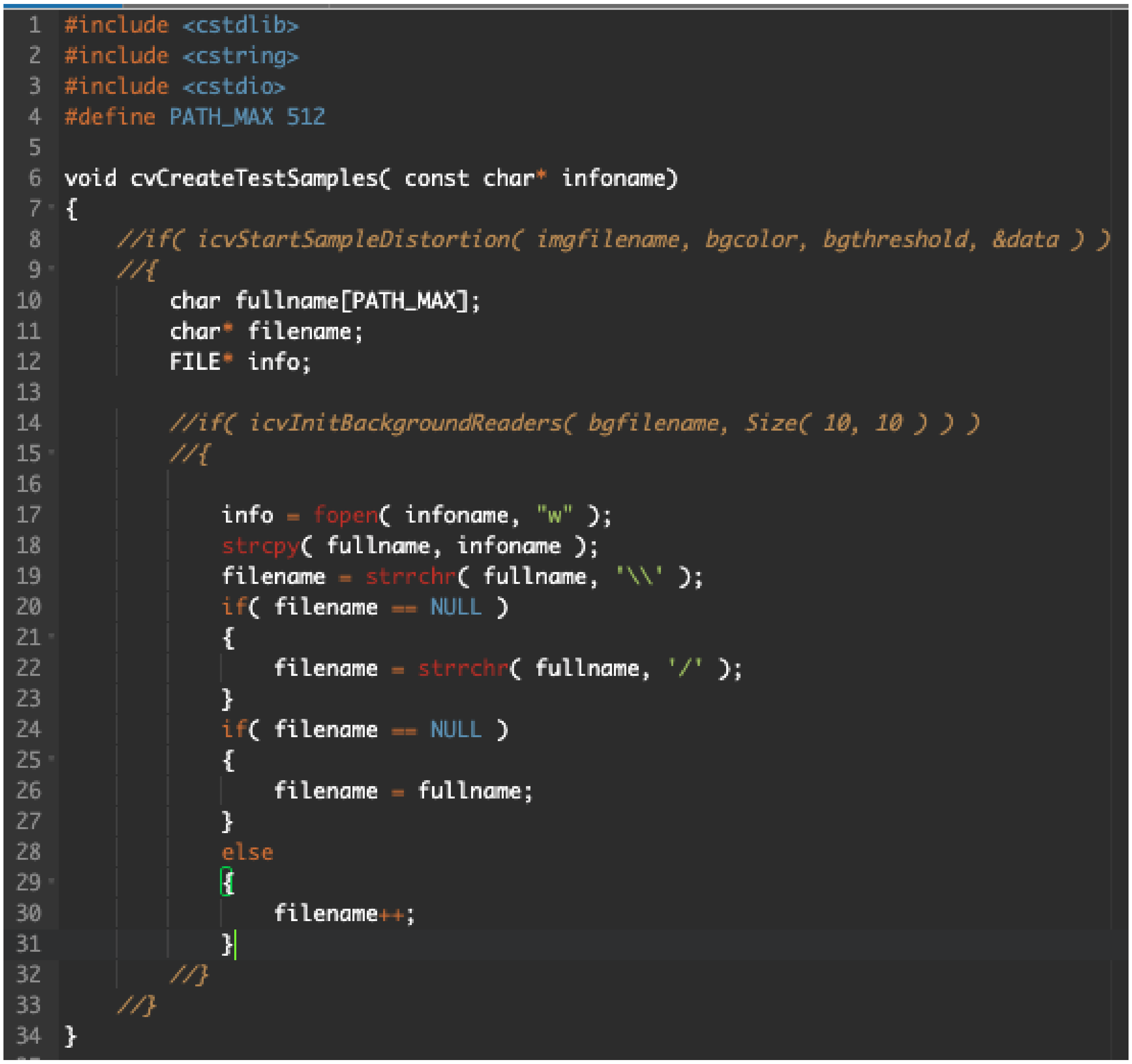


```
1116   void cvCreateTestSamples( const char* infoname,
1117                             const char* imgfilename, int bgcolor, int bgthreshold,
1118                             const char* bgfilename, int count,
1119                             int invert, int maxintensitydev,
1120                             double maxxangle, double maxyangle, double maxzangle,
1121                             int showsamples,
1122                             int winwidth, int winheight, double maxscale )
1123   {
```

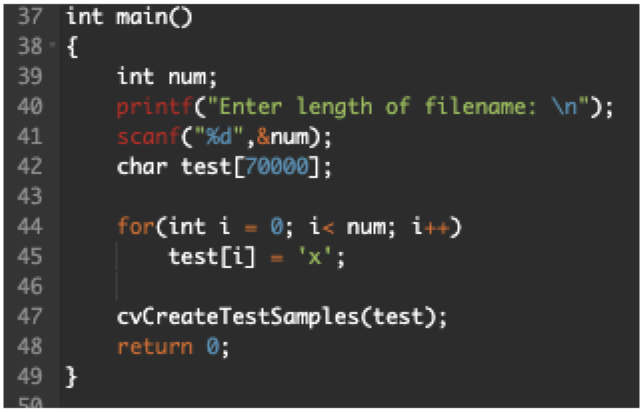


```
    char fullname[PATH_MAX];
        strcpy( fullname, infoname );
        filename = strrchr( fullname, '\\' );
        if( filename == NULL )
        {
            filename = strrchr( fullname, '/' );
        }
        if( filename == NULL )
        {
            filename = fullname;
        }
```

- **Category of attack:** Spatial memory (buffer overflow)
- **Reproducing process:** For the reproduction of the vulnerability, we separated the function which caused the vulnerability and compiled it separately. In doing so, we kept only the variables which were relevant to the vulnerability to avoid compiling errors. We then passed inputs to the reproduced function to expose the vulnerability.
  As shown below, the vulnerability is discovered once an input greater than the size of the destination buffer is passed to the function.

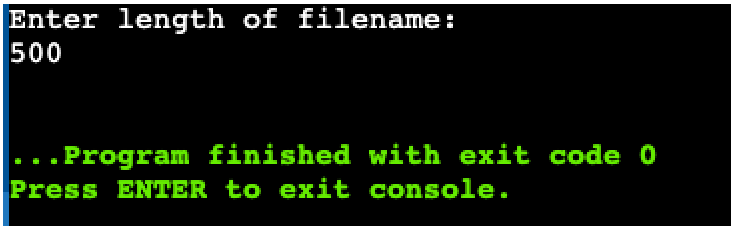


```
#include <cstdlib>
#include <cstring>
#include <cstdio>
#define PATH_MAX 512

void cvCreateTestSamples( const char* infoname)
{
    //if( icvStartSampleDistortion( imgfilename, bgcolor, bgthreshold, &data ) )
    //{
        char fullname[PATH_MAX];
        char* filename;
        FILE* info;

        //if( icvInitBackgroundReaders( bgfilename, Size( 10, 10 ) ) )
        //{

            info = fopen( infoname, "w" );
            strcpy( fullname, infoname );
            filename = strrchr( fullname, '\\' );
            if( filename == NULL )
            {
                filename = strrchr( fullname, '/' );
            }
            if( filename == NULL )
            {
                filename = fullname;
            }
            else
            {
                filename++;
            }
        //}
    //}
}
```



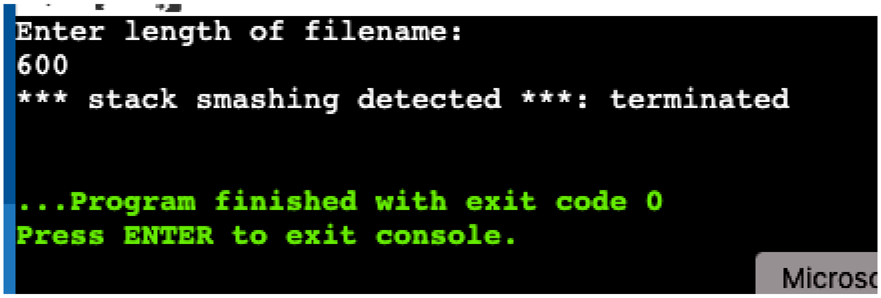
```
int main()
{
    int num;
    printf("Enter length of filename: \n");
    scanf("%d",&num);
    char test[70000];

    for(int i = 0; i< num; i++)
        test[i] = 'x';

    cvCreateTestSamples(test);
    return 0;
}
```

Output when the input is within the bound:



```
Enter length of filename:
500


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when the stack smashing happens:



```
Enter length of filename:
600
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

- **Mitigation:**

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

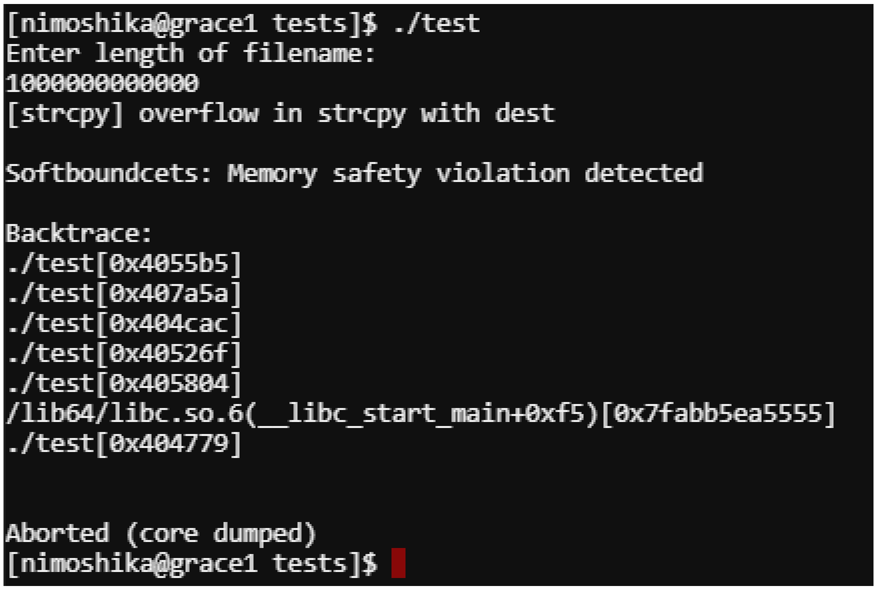
- **Demonstration Of Mitigation:**

The following steps were followed for the demonstration:

```
31.mkdir build; cd build
32.make -j8
33.export PATH=<git_repo>/llvm-38/build/bin:$PATH
34.make
35.clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
```

```
36../test
```

Below is the screenshot of the mitigation when the input size is out of bound. The error was discovered and it was aborted, which protects the program.

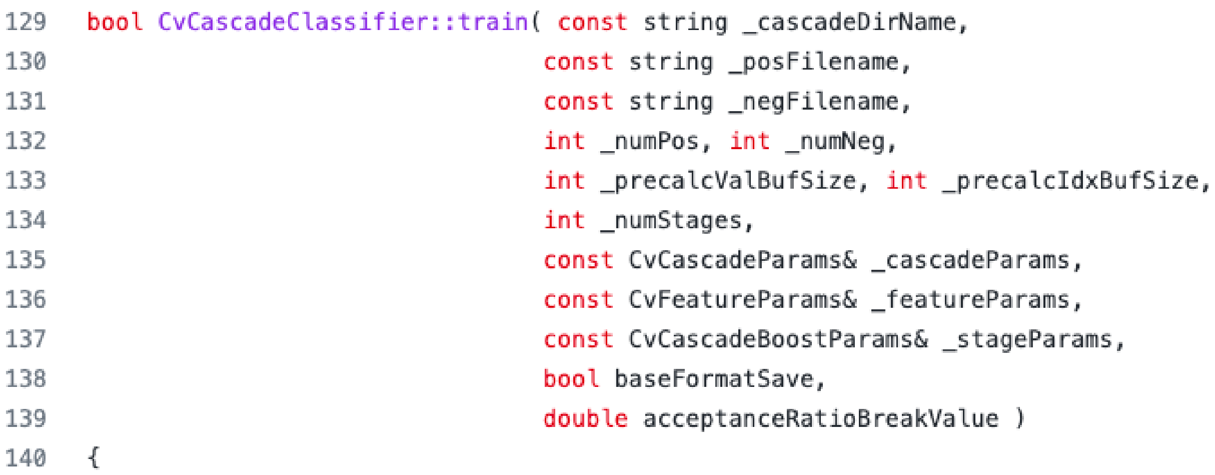


```
[nimoshika@grace1 tests]$ ./test
Enter length of filename:
1000000000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x4055b5]
./test[0x407a5a]
./test[0x404cac]
./test[0x40526f]
./test[0x405804]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7fabb5ea5555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

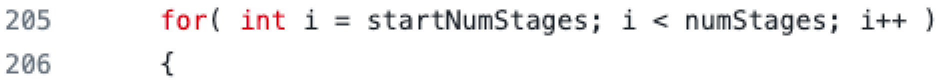
9) **Vulnerability Description:**

A buffer overflow happens when a program tries to write more data to a fixed length buffer beyond the buffer's capacity. This overwrites data in adjacent memory locations. This in turn can corrupt data in that adjacent space. This can corrupt data and cause system crashes. This also creates an opportunity for the attacker to execute arbitrary code, which is a vulnerability.

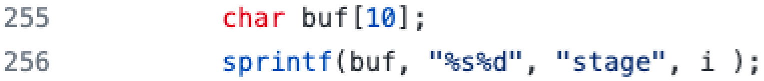
The vulnerability was detected through flawfinder. In the Github directory file opencv/apps/traincascade/cascadeclassifier.cpp (screenshot below), in line 256 inside the for loop, when integer "i" is greater than 10 characters (size of buf), a buffer overflows occurs.



```
bool CvCascadeClassifier::train( const string _cascadeDirName,
                                const string _posFilename,
                                const string _negFilename,
                                int _numPos, int _numNeg,
                                int _precalcValBufSize, int _precalcIdxBufSize,
                                int _numStages,
                                const CvCascadeParams& _cascadeParams,
                                const CvFeatureParams& _featureParams,
                                const CvCascadeBoostParams& _stageParams,
                                bool baseFormatSave,
                                double acceptanceRatioBreakValue )
{
```

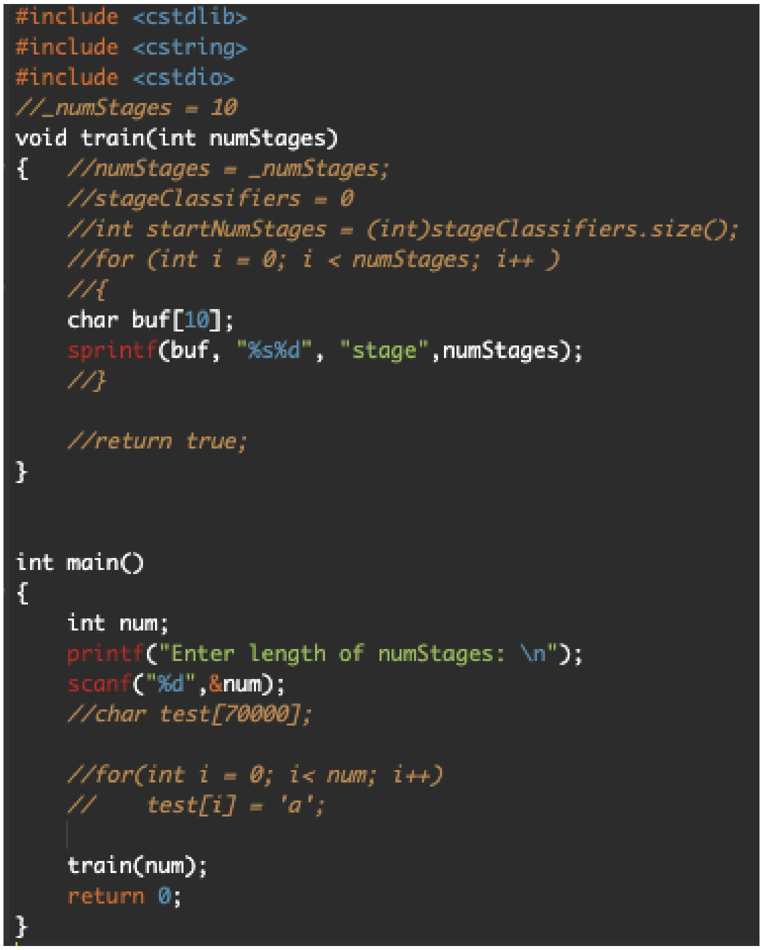


```
    for( int i = startNumStages; i < numStages; i++ )
    {
```



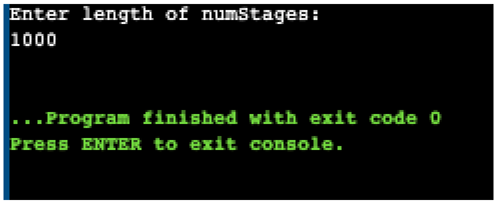
```
        char buf[10];
        sprintf(buf, "%s%d", "stage", i );
```

- **Category of attack:** Spatial memory (buffer overflow)
- **Reproducing process:** For the reproduction of the vulnerability, we separated the function which caused the vulnerability and compiled it separately. In doing so, we kept only the variables which were relevant to the vulnerability to avoid compiling errors. We then passed inputs to the reproduced function to expose the vulnerability.
  As shown below, the vulnerability is discovered once an input greater than the size of the destination buffer is passed to the function.



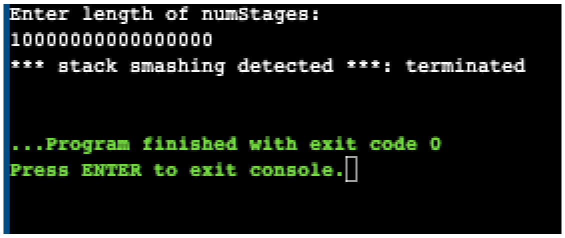
```
#include <cstdlib>
#include <cstring>
#include <cstdio>
//_numStages = 10
void train(int numStages)
{   //numStages = _numStages;
    //stageClassifiers = 0
    //int startNumStages = (int)stageClassifiers.size();
    //for (int i = 0; i < numStages; i++ )
    //{
    char buf[10];
    sprintf(buf, "%s%d", "stage",numStages);
    //}

    //return true;
}


int main()
{
    int num;
    printf("Enter length of numStages: \n");
    scanf("%d",&num);
    //char test[70000];

    //for(int i = 0; i< num; i++)
    //    test[i] = 'a';

    train(num);
    return 0;
}
```

Output when the input is within the bound:



```
Enter length of numStages:
1000


...Program finished with exit code 0
Press ENTER to exit console.
```

Output when the stack smashing happens:



```
Enter length of numStages:
10000000000000000
*** stack smashing detected ***: terminated


...Program finished with exit code 0
Press ENTER to exit console.
```

- **Mitigation:**

Strategy3: Detect the malicious use cases: Bounds Checking(Softbound):

Softbound checking was done using the reference - https://github.com/santoshn/softboundcets-34, where the reproduced code is tested under the test cases. The memory safety violation is flagged. This helps in terminating the program when vulnerability is detected. The program may get killed prematurely.

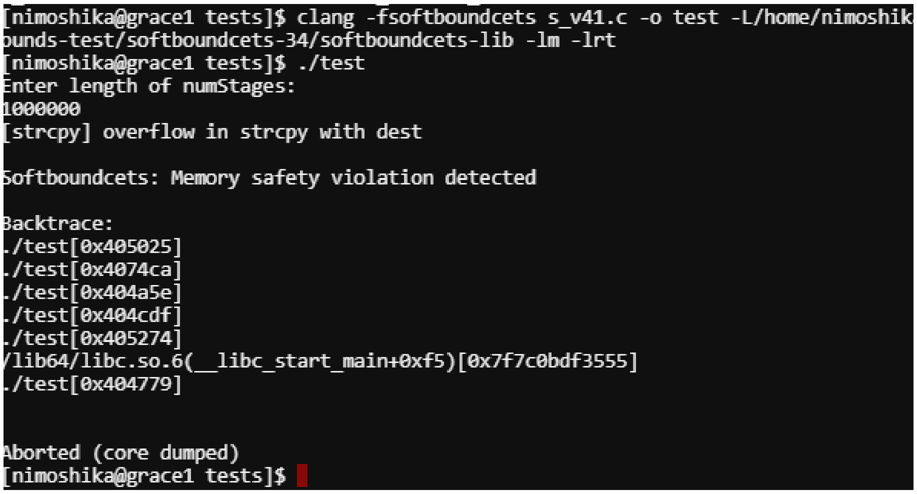
SoftBound and CETS enforces complete spatial and temporal safety for C through compile-time transformations, thus providing full memory safety for C. In SoftBound+CETs pointer based checking with bounds is used and metadata is identified with every pointer. Bound information for every pointer is recorded as disjoint metadata (source: https://people.cs.rutgers.edu/~sn349/softbound/).

- **Demonstration Of Mitigation:**

The following steps were followed for the demonstration:

```
37.mkdir build; cd build
38.make -j8
39.export PATH=<git_repo>/llvm-38/build/bin:$PATH
40.make
41.clang -fsoftboundcets test.c -o test -L<git_repo>/runtime -lm -lrt
   -lsoftboundcets_rt
42../test
```

Below is the screenshot of the mitigation when the input size is out of bound. The error was discovered and it was aborted, which protects the program.



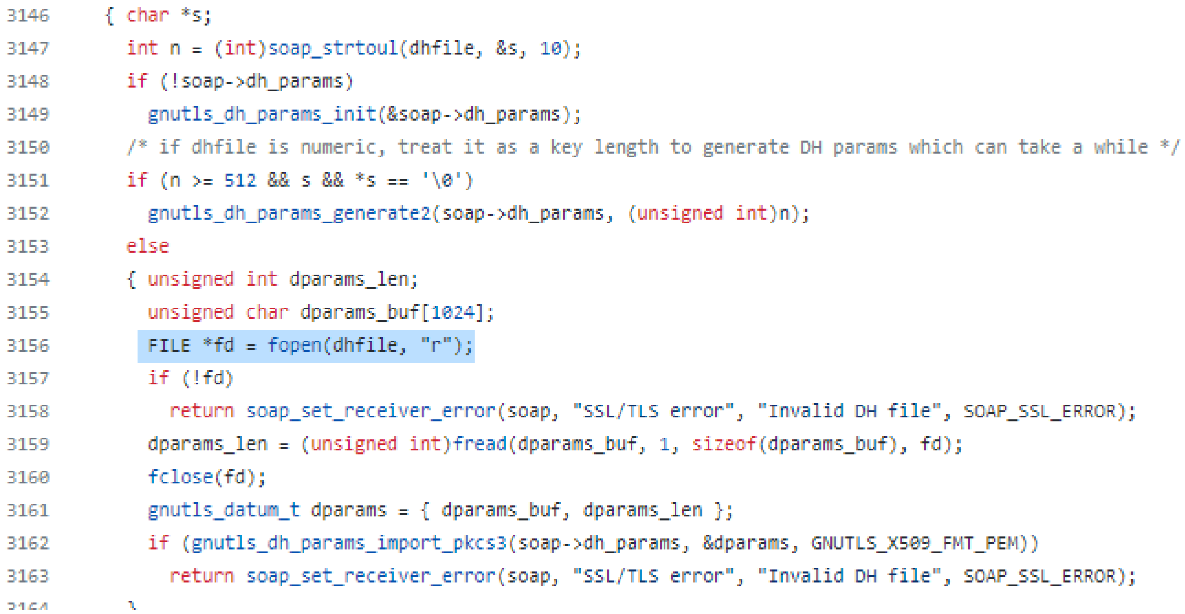
```
[nimoshika@grace1 tests]$ clang -fsoftboundcets s_v41.c -o test -L/home/nimoshik
ounds-test/softboundcets-34/softboundcets-lib -lm -lrt
[nimoshika@grace1 tests]$ ./test
Enter length of numStages:
1000000
[strcpy] overflow in strcpy with dest

Softboundcets: Memory safety violation detected

Backtrace:
./test[0x405025]
./test[0x4074ca]
./test[0x404a5e]
./test[0x404cdf]
./test[0x405274]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7f7c0bdf3555]
./test[0x404779]


Aborted (core dumped)
[nimoshika@grace1 tests]$
```

10) **Vulnerability Description:**

Time-to-check-t0-time-to-use is the most common form of vulnerability where the attacker uses the symbolic link to enter the code to read another file which might affect the confidentiality and integrity. When opening files ,an attacker redirects it (via symlinks), forces the opening of a special file type (e.g., device files), moves things around to create a race condition, control its ancestors, or change its contents.

In location, onvif-qt-server-client-master/onviflibs/OnvifDeviceIOLib/soap/stdsoap2.cpp:3156 “fopen” is used which tries to open the file and this area makes the attacker check the file, open it and make changes in the interpretation of the file name and let the victim read the secret file. Attacker uses symlink.

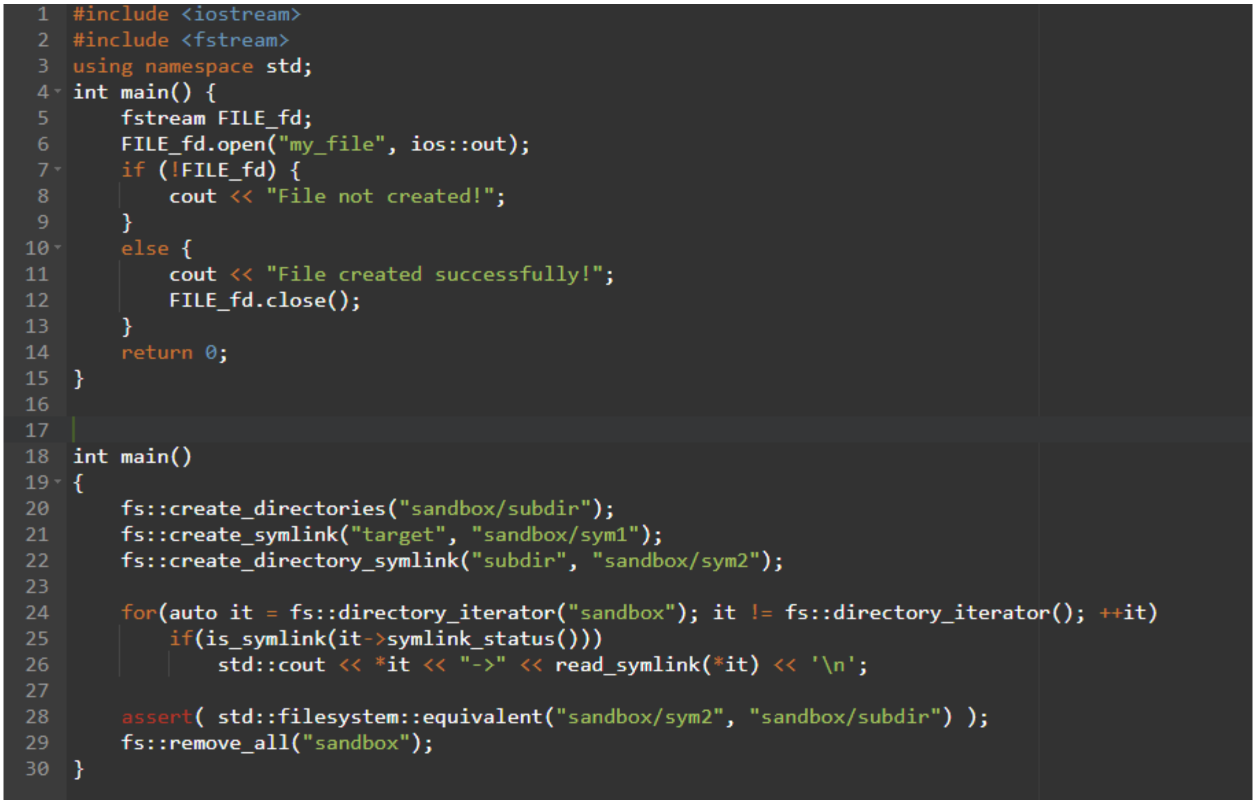
Category: Concurrency Attacks(Time-To-Check-To-Time-To-Use)



```
3146    { char *s;
3147      int n = (int)soap_strtoul(dhfile, &s, 10);
3148      if (!soap->dh_params)
3149        gnutls_dh_params_init(&soap->dh_params);
3150      /* if dhfile is numeric, treat it as a key length to generate DH params which can take a while */
3151      if (n >= 512 && s && *s == '\0')
3152        gnutls_dh_params_generate2(soap->dh_params, (unsigned int)n);
3153      else
3154      { unsigned int dparams_len;
3155        unsigned char dparams_buf[1024];
3156        FILE *fd = fopen(dhfile, "r");
3157        if (!fd)
3158          return soap_set_receiver_error(soap, "SSL/TLS error", "Invalid DH file", SOAP_SSL_ERROR);
3159        dparams_len = (unsigned int)fread(dparams_buf, 1, sizeof(dparams_buf), fd);
3160        fclose(fd);
3161        gnutls_datum_t dparams = { dparams_buf, dparams_len };
3162        if (gnutls_dh_params_import_pkcs3(soap->dh_params, &dparams, GNUTLS_X509_FMT_PEM))
3163          return soap_set_receiver_error(soap, "SSL/TLS error", "Invalid DH file", SOAP_SSL_ERROR);
3164      }
```

Reproducing Process:

Symbolic link is created and added to the existing directory. It is added in such a way that when a log file is generated for the particular directory/folder, the symbolic link gets created automatically. While the program is executing, the contents of the file would be leaked. A sample function of how a file read close is done and a symlink is created is shown below.



```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    fstream FILE_fd;
    FILE_fd.open("my_file", ios::out);
    if (!FILE_fd) {
        cout << "File not created!";
    }
    else {
        cout << "File created successfully!";
        FILE_fd.close();
    }
    return 0;
}


int main()
{
    fs::create_directories("sandbox/subdir");
    fs::create_symlink("target", "sandbox/sym1");
    fs::create_directory_symlink("subdir", "sandbox/sym2");

    for(auto it = fs::directory_iterator("sandbox"); it != fs::directory_iterator(); ++it)
        if(is_symlink(it->symlink_status()))
            std::cout << *it << "->" << read_symlink(*it) << '\n';

    assert( std::filesystem::equivalent("sandbox/sym2", "sandbox/subdir") );
    fs::remove_all("sandbox");
}
```

<u>Mitigation:</u>

<u>Strategy3: Detect the malicious use cases: Ptrace : Process Trace</u>

In this method, one process takes the responsibility of controlling and monitoring the other process's change in memory and registers. It is useful in implementing the breakpoint debugging and system call tracing. Ptrace is generally called by using the processID. Once called, it becomes the child of the other process. Once it attaches to the process and starts it's tracing, it examines the pointers and detaches.

```
ptrace(PTRACE_ATTACH, traced_process,
           NULL, NULL);
```

```
ptrace(PTRACE_DETACH, traced_process,
           NULL, NULL);
```

The above mentioned tracing calls are used to attach and detach in between the already existing code. when a Ptrace is called , the kernel sets up the flag to denote that it is being traced. The child process is stopped and the parent does the tracing. Once done, it wakes up the child.

Ptrace is very helpful in flagging the concurrency (TTCTTO) attacks as it can examine and modify the dynamic program.