Exploring the STL: Owning erase( )
Draft - 20 June 2009
Chris.Rohlf@gmail.com

All examples were created on x86 Ubuntu Linux 9.04 (g++ version 4.3.3 / Ubuntu 4.3.3-5ubuntu4)

If you have ever written C++ code before you have no doubt used STL containers. They make complex data structures very easy to use. There are a bunch of methods to these containers that allow you to store, retrieve and remove data from them. Some of these methods have interesting side effects when an attacker can influence them. The erase method is one such case. These containers, while very convenient, are not magic. They are merely heap chunks and pointers. Always keep this in mind when you find and exploit STL bugs. If you write C++ applications that use these containers, be aware, they can be easily abused into creating exploitable situations.

The erase method is available to a few different STL containers. This article will focus on the vector, as it is easily understood and used often in the real world because of its simplicity and efficiency. A vector is a dynamic array that holds contiguous elements. When you create a vector you define what type of vector it is. In my examples we create a vector to hold int's, but the same concepts apply to vectors holding any data type.

The erase method basically takes two forms. You can either give it a single index into the container and it will erase that element or you can give it a range of positions and it will delete all the elements within that range. This article specifically focuses on passing erase an attacker influenced range of elements to destroy. The Certs secure coding standards [1] begins to touch on the subject of invalid iterator ranges, but labels their 'undefined behavior' as equivalent to a buffer overflow. This is true, however it can be more then that depending on implementation, as I will demonstrate.

But before we dive into the details of attacking erase we need a quick intro into how erase works under the hood:

When you call 'container.erase(start, end)'. 'start' and 'end' are iterators (pointers) that specify the range of elements you want removed. For example your start iterator may point to the 2nd element in the vector and your end iterator may point to the 5th. In this initial call to erase the STL checks if the end iterator equals the last element in the container. If it does not then the STL calls a macro _GLIBCXX_MOVE3, which is really just a std::move() or std::copy(), which further boils down to a call to memmove. You can verify this using ltrace and a simple program that calls erase on a container. The macro looks like this pseudo code:

```
erase(__first, __last) {
    _GLIBCXX_MOVE3(__last, end(), __first);
}
```

Lets say you have a vector that holds 9 elements each with a letter of the alphabet:

```
[0]a [1]b [2]c [3]d [4]e [5]f [6]g [7]h [8]i [9]j
```

Now we want to delete 'c' through 'e'. We call erase like this:

```
vector.erase(iterator_pointing_to_element_2, iterator_pointing_to_element_5)
```

Now the call to _GLIBCXX_MOVE3 would look like this:

```
_GLIBCXX_MOVE3(5, 9, 2)
```

Which would copy the range of 5 to 9 (f -> j), the elements we want to keep, starting at element 2. Which would leave us with:

[0]a [1]b [2]f [3]g [4]h [5]i [6]j

Simple right? Its basically just a memmove. After the call to memmove erase() is going to call _M_erase_at_end(). This function essentially takes a single pointer as its only argument and calls std::_Destroy on this pointer. Then a destructor for each of the discarded elements will be called, this may be an interesting area to explore in the future but I will leave that up to the reader for now.

Let me introduce a small sample program, erase.cpp. This program creates three vectors 'bec', 'vec' and 'dec'. It also takes two command line arguments which allow us to control the value of the iterators passed to erase(). A note to the reader, I actually cheat here and just pass these command line arguments as additions to vector.begin() for simplicty, but it has the same effect.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <vector>

using namespace std;

ostream &operator << (ostream &os, const vector<int> &coll)
{
  os << "@ " << &coll[0] << " | " << coll.size() << " ";
  vector<int>::const_iterator start = coll.begin();
  for(int i=0; i<coll.size(); i++, start++) {
    os << "[" << i << "] " << hex << *start << dec << " ";
  }
  return os;
}

int main(int argc, char *argv[])
{
  vector<int> bec, vec, dec;

  int b = atoi(argv[1]);
  int c = atoi(argv[2]);

  char *m = (char *)malloc(4096);
  memset(m, 0x61, 4096);
  free(m);

  bec.assign(10, 0x42424242);
  vec.assign(10, 0x43434343);
  dec.assign(10, 0x44444444);

  cout << "bec " << &bec[0] << endl << "vec " << &vec[0] << endl << "dec " << &dec[0] << endl;

  cout << endl << "bec " << bec << endl;
  cout << endl << "vec " << vec << endl;
  cout << endl << "dec " << dec << endl;
```

```
    vec.erase(vec.begin()+b, vec.begin()+c);

    cout  << endl << "erase called!" << endl;

    cout << endl << "bec " << bec << endl;
    cout << endl << "vec " << vec << endl;
    cout << endl << "dec " << dec << endl;

    return 0;
}
```

You can compile this program with g++

$ g++ -o erase erase.cpp -ggdb

First we allocate a large chunk of memory (4096 bytes) and fill it with 0x61, then we free it. This is to mark the heap with an easily recognizable pattern so that we can detect any uninitialized bytes of memory within our vectors. This step is not necessary but can be helpful when staring at a hexadecimal dump of the heap contents in a debugger. We then fill 3 the vectors bec, vec, and dec with 0x42, 0x43 and 0x44 respectively. All three are (sizeof(int)*10) in size (40 bytes). This should make it easy to spot their boundaries in our debugger. If we want to erase elements 2 through 9 in our 'vec' vector then we supply the arguments as shown below.

```
$./erase 2 9
bec 0x9cd6008
vec 0x9cd6038
dec 0x9cd6068

bec @ 0x9cd6008 10 | [0] 42424242 [1] 42424242 [2] 42424242 [3] 42424242 [4] 42424242 [5] 42424242 [6] 42424242 [7]
42424242 [8] 42424242 [9] 42424242

vec @ 0x9cd6038 10 | [0] 43434343 [1] 43434343 [2] 43434343 [3] 43434343 [4] 43434343 [5] 43434343 [6] 43434343 [7]
43434343 [8] 43434343 [9] 43434343

dec @ 0x9cd6068 10 | [0] 44444444 [1] 44444444 [2] 44444444 [3] 44444444 [4] 44444444 [5] 44444444 [6] 44444444 [7]
44444444 [8] 44444444 [9] 44444444

erase called!

bec @ 0x9cd6008 10 | [0] 42424242 [1] 42424242 [2] 42424242 [3] 42424242 [4] 42424242 [5] 42424242 [6] 42424242 [7]
42424242 [8] 42424242 [9] 42424242

vec @ 0x9cd6038 3 | [0] 43434343 [1] 43434343 [2] 43434343

dec @ 0x9cd6068 10 | [0] 44444444 [1] 44444444 [2] 44444444 [3] 44444444 [4] 44444444 [5] 44444444 [6] 44444444 [7]
44444444 [8] 44444444 [9] 44444444
```

Our program printed out the contents of each vector before and after the call to erase().

Remember, erase is calling memmove to move the last element back towards the beginning of the vector. So what will happen if we give it a starting index far larger then the size of the vector itself? Lets try it out ...

```
$./erase 50 0
```

bec 0x83ba008
vec 0x83ba038
dec 0x83ba068

bec @ 0x83ba008 10 | [0] 42424242 [1] 42424242 [2] 42424242 [3] 42424242 [4] 42424242 [5] 42424242 [6] 42424242 [7] 42424242 [8] 42424242 [9] 42424242

vec @ 0x83ba038 10 | [0] 43434343 [1] 43434343 [2] 43434343 [3] 43434343 [4] 43434343 [5] 43434343 [6] 43434343 [7] 43434343 [8] 43434343 [9] 43434343

dec @ 0x83ba068 10 | [0] 44444444 [1] 44444444 [2] 44444444 [3] 44444444 [4] 44444444 [5] 44444444 [6] 44444444 [7] 44444444 [8] 44444444 [9] 44444444

erase called!

bec @ 0x83ba008 10 | [0] 42424242 [1] 42424242 [2] 42424242 [3] 42424242 [4] 42424242 [5] 42424242 [6] 42424242 [7] 42424242 [8] 42424242 [9] 42424242

vec @ 0x83ba038 60 | [0] 43434343 [1] 43434343 [2] 43434343 [3] 43434343 [4] 43434343 [5] 43434343 [6] 43434343 [7] 43434343 [8] 43434343 [9] 43434343 [10] 3616172 [11] 31 [12] 44444444 [13] 44444444 [14] 44444444 [15] 44444444 [16] 44444444 [17] 44444444 [18] 44444444 [19] 44444444 [20] 44444444 [21] 44444444 [22] 3616178 [23] 20f71 [24] 61616161 [25] 61616161 [26] 61616161 [27] 61616161 [28] 61616161 [29] 61616161 [30] 61616161 [31] 61616161 [32] 61616161 [33] 61616161 [34] 61616161 [35] 61616161 [36] 61616161 [37] 61616161 [38] 61616161 [39] 61616161 [40] 61616161 [41] 61616161 [42] 61616161 [43] 61616161 [44] 61616161 [45] 61616161 [46] 61616161 [47] 61616161 [48] 61616161 [49] 61616161 [50] 43434343 [51] 43434343 [52] 43434343 [53] 43434343 [54] 43434343 [55] 43434343 [56] 43434343 [57] 43434343 [58] 43434343 [59] 43434343

dec @ 0x83ba068 10 | [0] 44444444 [1] 44444444 [2] 44444444 [3] 44444444 [4] 44444444 [5] 44444444 [6] 44444444 [7] 44444444 [8] 44444444 [9] 44444444
*** glibc detected *** ./erase: free(): invalid pointer: 0x83ba068

Here's what happened. We gave the start iterator a value of vec.begin()+50 and an end iterator of vec.begin()+0. We essentially tricked it into reading memory beyond the boundary of our 'vec' vector. This is obvious by looking at the contents of vec that was printed out, and its size of 60! Note the size of 60 as it is a significant detail. The GLibc error message is because we overwrote some heap chunk meta data in the process, but i'll explain that more below. Lets run this in GDB and inspect the heap chunks directly. If you run our sample program 'erase' with different values you will get different results, try it out.

Here's a deeper look at what's happening. We are going to run erase.cpp with the arguments '20' and '0' and we are going to break at line 40 and line 45 which is before and after the call to erase().

(gdb) r 20 0
Starting program: /erase 20 0
bec 0x8920008
vec 0x8920038
dec 0x8920068

bec @ 0x8920008 10 | [0] 42424242 [1] 42424242 [2] 42424242 [3] 42424242 [4] 42424242 [5] 42424242 [6] 42424242 [7] 42424242 [8] 42424242 [9] 42424242

vec @ 0x8920038 10 | [0] 43434343 [1] 43434343 [2] 43434343 [3] 43434343 [4] 43434343 [5] 43434343 [6] 43434343 [7] 43434343 [8] 43434343 [9] 43434343

dec @ 0x8920068 10 | [0] 44444444 [1] 44444444 [2] 44444444 [3] 44444444 [4] 44444444 [5] 44444444 [6] 44444444 [7]

44444444 [8] 44444444 [9] 44444444

Breakpoint 1, main (argc=Cannot access memory at address 0xa) at erase.cpp:40
40 vec.erase(vec.begin()+b, vec.begin()+c);
(gdb) x/50x 0x8920008-8 <- The -8 is so we can see the beginning of the chunk at 0x8920008 (start of the heap)
0x8920000: 0x00000000 0x00000031 0x42424242 0x42424242 <- Start of allocated chunk at 0x8920000 (bec)
0x8920010: 0x42424242 0x42424242 0x42424242 0x42424242
0x8920020: 0x42424242 0x42424242 0x42424242 0x42424242
0x8920030: 0x03616140 0x00000031 0x43434343 0x43434343 <- Start of allocated chunk at 0x8920030 (vec)
0x8920040: 0x43434343 0x43434343 0x43434343 0x43434343
0x8920050: 0x43434343 0x43434343 0x43434343 0x43434343
0x8920060: 0x03616146 0x00000031 0x44444444 0x44444444 <- Start of allocated chunk at 0x8920060 (dec)
0x8920070: 0x44444444 0x44444444 0x44444444 0x44444444
0x8920080: 0x44444444 0x44444444 0x44444444 0x44444444
0x8920090: 0x0361614c 0x00020f71 0x61616161 0x61616161 <- Start of free chunk at 0x8920090
0x89200a0: 0x61616161 0x61616161 0x61616161 0x61616161 <- 0x61's are from the earlier buffer returned by malloc
0x89200b0: 0x61616161 0x61616161 0x61616161 0x61616161
0x89200c0: 0x61616161 0x61616161
(gdb) c
Continuing.

erase called! <- erase has been called

Breakpoint 2, main (argc=Cannot access memory at address 0xa) at erase.cpp:45
45 cout << endl << "vec " << vec << endl;
(gdb) x/50x 0x8920008-8
0x8920000: 0x00000000 0x00000031 0x42424242 0x42424242
0x8920010: 0x42424242 0x42424242 0x42424242 0x42424242
0x8920020: 0x42424242 0x42424242 0x42424242 0x42424242
0x8920030: 0x03616140 0x00000031 0x43434343 0x43434343 <- 'vec' starts at 0x8920038 and is now 30 bytes in size (look below for size output)
0x8920040: 0x43434343 0x43434343 0x43434343 0x43434343
0x8920050: 0x43434343 0x43434343 0x43434343 0x43434343
0x8920060: 0x03616146 0x00000031 0x44444444 0x44444444 <- The heap chunk at 0x8920060 (dec) is now within the 'vec' vector
0x8920070: 0x44444444 0x44444444 0x44444444 0x44444444
0x8920080: 0x44444444 0x44444444 0x43434343 0x43434343
0x8920090: 0x43434343 0x43434343 0x43434343 0x43434343 <- Our free chunk header was overwritten with the contents of 'vec'
0x89200a0: 0x43434343 0x43434343 0x43434343 0x43434343 <- Still within vec boundary
0x89200b0: 0x61616161 0x61616161 0x61616161 0x61616161
0x89200c0: 0x61616161 0x61616161
(gdb) c
Continuing.

bec @ 0x8920008 10 | [0] 42424242 [1] 42424242 [2] 42424242 [3] 42424242 [4] 42424242 [5] 42424242 [6] 42424242 [7] 42424242 [8] 42424242 [9] 42424242

vec @ 0x8920038 30 | [0] 43434343 [1] 43434343 [2] 43434343 [3] 43434343 [4] 43434343 [5] 43434343 [6] 43434343 [7] 43434343 [8] 43434343 [9] 43434343 [10] 3616146 [11] 31 [12] 44444444 [13] 44444444 [14] 44444444 [15] 44444444 [16] 44444444 [17] 44444444 [18] 44444444 [19] 44444444 [20] 43434343 [21] 43434343 [22] 43434343 [23] 43434343 [24] 43434343 [25] 43434343 [26] 43434343 [27] 43434343 [28] 43434343 [29] 43434343

```
dec @ 0x8920068 10 | [0] 44444444 [1] 44444444 [2] 44444444 [3] 44444444 [4] 44444444 [5] 44444444 [6] 44444444 [7]
44444444 [8] 43434343 [9] 43434343
*** glibc detected *** /erase: free(): invalid pointer: 0x08920068 ***
*** glibc detected *** /erase: malloc: top chunk is corrupt: 0x08920090 ***
(gdb)
```

Does element 10 or 11 of 'vec' stand out to you? They should, they are the chunk header of the buffer containing the 'dec' vector.

The 'dec' vector, and its associated heap meta data, is now located within the 'vec' vector and the STL is none the wiser. This is proven by taking the address of vec's first element at 0x8920038 and adding its size, which was printed out above, (30 * 4) + 0x8920038 = 0x89200b0. However the address of the first element in dec is at 0x8920068 and if we apply the same math its final element should be at 0x8920068 + (10 * 4) = 0x8920090 which is where the chunk header for the next free chunk was located at.

This puts 'dec' well within 'vec', and assuming we can make further writes within 'vec', allows us to control the entire contents of 'dec' including its associated heap chunk meta data. this can be confirmed by printing 'vec' and 'dec' in GDB:

```
(gdb) print vec
$5 = {<std::_Vector_base<int, std::allocator<int> >> = {_M_impl = {<std::allocator<int>> =
{<__gnu_cxx::new_allocator<int>> = {<No data fields>}, <No data fields>}, _M_start = 0x8920038,
_M_finish = 0x89200b0, _M_end_of_storage = 0x8920060}}, <No data fields>}
(gdb) print dec
$6 = {<std::_Vector_base<int, std::allocator<int> >> = {_M_impl = {<std::allocator<int>> =
{<__gnu_cxx::new_allocator<int>> = {<No data fields>}, <No data fields>}, _M_start = 0x8920068,
_M_finish = 0x8920090, _M_end_of_storage = 0x8920090}}, <No data fields>}
```

Notice the _M_start and _M_finish values. This gives us a lot of leverage when writing an exploit. Consider the following pseudo code:

```
// Make sure we write user_supplied_stuff
// within the bounds of vec

pos_1 = user_controlled;

if(pos_1 > vec.begin() && pos_1 < vec.end())    {
   &vec[pos_1] = evil_content_to_overwrite_a_function_ptr_in_dec;
}
```

The above range check is valid but allows us to write data directly into the 'dec' vector, including its associated heap chunk data. There is also the possibility that we already control the contents of 'dec' but now it will be used as a part of what should be in 'vec'. This type of scenario may allow for even more leverage if we want to corrupt or append to the contents of 'vec' but can't overflow 'bec'. We simply fill 'dec' with the desired contents and then invalidate the size of 'vec' via an attacker influenced call to erase().

Exploiting this situation to exploit arbitrary code is possible in a number of ways. In our example above we can modify the vec elements [10] and [11] to control the heap chunk meta data of 'dec'. Fortunately its not 2002 and GLibc is going to check for any inconsistencies, as seen above. However there is still hope for the exploit writer because if we can control the offsets passed to erase we can influence what bytes become sucked into the vector whose contents we control. There is plenty of other good stuff to overwrite on the heap such as function pointers, passwords etc... In fact today it is much easier to attack the heap contents and not the heap itself. This fact makes the circumstances of this type of vulnerability particularly

interesting, as it won't instantly corrupt your heap, and may allow for data leakage first, followed by the opportunity to overwrite that data. The only potential problem here is that the size of the 'vec' vector has been modified, which may lead conditions where the target crashes itself by over writing critical data within 'dec'. Other then that an exploit writer could not ask for better conditions.

The example demonstrated here is quite simple and you probably wont be able to influence both iterators in the real world, but having influence over either one may give you something to play with. The possibility of an integer over/underflow may also arise, in which case you may be able to read further into the heap or have control over larger amounts of memory corruption by gaining more control over iterator positions. In the real world you may be able to influence iterators by means of file structure fields or in the data fields of a particular network protocol. Plenty of vulnerabilities have been discovered which are the direct result of an attacker controlled pointer, and this is the same concept with an STL twist.

While the examples in this article focused on Linux and libstdc, the same concepts apply to OS X (they use the same templates). A similar side effect of attacker controlled iterators also works on Windows XP on code compiled with VS Express 2008. The attack is slightly modified and doesn't allow as much control. The start iterator must be (<= vec.size) and the end iterator must be (<= start iterator). Using our example above compiled on Windows, give it the arguments '10 0'. This allows you to overwrite the contents of dec, but does not allow you to encapsulate dec within vec. What happens here is closer to what the Certs secure coding standards mention [1]. Buffer overflow type conditions do occur, but in a limited way.

The topic covered here may be a well known thing to experienced C++ developers but the STL is an interesting beast, and the security research community has not given it the attention it deserves. Using tainted iterators has long been known to be a bad thing, but exploiting those situations has barely been touched. There are plenty of other untouched methods that perform memory copies under the hood, all of which are waiting to be explored and broken.

[1] https://www.securecoding.cert.org/confluence/display/cplusplus/ARR34-CPP.+Use+Valid+Iterator+Ranges