Lab: Code Safety: Memory

Software, hardware, and systems should do everything that they are intended to do and nothing more. The “nothing more” part of that statement is what is meant by safety. If, say code, is allowed to do more than what is intended an attacker just might be able to exploit unintended execution of given code and perform some malicious task or compromise privacy, for example. Numerous programming languages require great care to develop a safe product. The C/C++ family is most notorious in allowing a developer to create exploitable code but other popular languages do so as well. Three classes of safety are of particular importance because they have been most responsible for unintended execution. These are: memory safety, thread safety, and type safety. This lab is concerned with memory safety. By memory safety we mean:

1. No variable in a stack references an uninitialized object. Thus no variable can dereference a null object.
2. No variable in a stack references an object whose lifetime is shorter. Thus, a reference to space that has been reclaimed by the operating system is prohibited.
3. All references respect permissions

Consider the following C++ code, provided in file `mem-unsafe-1.cc`, for illustration:

```
#include <iostream>
using namespace std;

class Object {
 private:
   int number;

 public:
   Object (int n) { number = n; }
   int getNumber () { return number; }
 };

int main () {
   cout << "first object created, ";
   Object *object_1 = new Object(10);
   cout << "\tfirst's number: " << object_1->getNumber() << "\n";
   delete object_1;
   cout << "first object deleted, ";
   cout << "\tfirst's number: " << object_1->getNumber() << "\n";
   Object *object_2 = new Object(13);
   cout << "second object created, ";
   cout << "\tfirst's number: " << object_1->getNumber() << "\n";
}
```

Execution of this code is as follows: The operating system assigns to pointer `object_1` space for a data object of type `Object`, with variable `number` initialized to 10. Then delete is applied to `object_1` which means `object_1` remains alive while the object it was assigned is no longer alive, in violation of point 2. above. Then pointer `object_2` is assigned to a new object of class `Object` with variable `number` initialized to 13. Finally, `Object`’s `getNumber` is accessed through `object_1`. This is allowed in C++ and the result is 13. The reason for this is the operating system reused the memory it reclaimed from the `delete` applied to `object_1` to create on object that it assigned to `object_2`. Since `object_1` was still pointing to that space, the number 13 was returned by `object_1->getNumber();` But such an operation is undefined and could lead to serious memory leakage in practice. Compile and run this code to get the following:

```
first object created,   first's number: 10
first object deleted,   first's number: 1431655787
second object created,  first's number: 13
```

Now, consider the following code, provided in file `mem-unsafe-2.cc`:

```
#include <iostream>
using namespace std;

class A {
private:
   int a,b,c;

public:
   A (int x) { a = x; b=x+20; c=x+31; }
   int get() { return a; }
};

int main () {
   A *a = new A(125);
   cout << a->get() << "\n";
   /** cout << a->a << "\n"; **/ /* compiler error - private access!! */
   *((int*)a+2) = 876;
   cout << *(int*)a << "\n";
   cout << *((int*)a+1) << "\n";
   cout << *((int*)a+2) << "\n";
}
```

Pointer `a` is assigned an object of class `A` with private variable `a` set to 125, private variable `b` set to 145 and private variable `c` set to 156. Method `get` is used to display the value of `a`. There is no public method for accessing private variables `b` and `c`. However, the address of variable `a` is the address of the object of class `A`. Since `a`,`b`, and `c` are of type `int`, and they appear consecutively in memory in the order `a`, `b`, `c`, the address of `b` is `(int*)a+1` and the address of `c` is `(int*)a+2`. Thus `*((int*)a+2) = 876;` changes the value of `c` to 876 and this change is illustrated using `cout << *((int*)a+2) << “\n”;` Such unexpected behavior is a violation of 3. above.

Now, consider the following code, provided in file `mem-unsafe-3.cc`:

```
#include <iostream>
using namespace std;

class A {
   public:
      A (int n) { number = n; }
      int number;
};

int main () {
   void *a = (void*)malloc(sizeof(A));
   cout << ((A*)a)->number << "\n";
}
```

A pointer of type `void*` is created and assigned, via `malloc`, a chunk of space at least equal in size to the space used by objects of class `A`. Then pointer `a` is cast as a pointer to an object of class `A` and `number` is de-referenced. But there is no such object. This is a violation of 1. above. Running this code may output a 0 but that depends on the compiler. It may even be possible that some sensitive data that remains in memory after some prior execution is provided by the `malloc` call.

The space allocated to a process stack has numerous protections to prevent exploitation. Examples are canaries to prevent stack overflows from corrupting stack values such as return addresses, no execute on the stack to prevent an attacker from filling a buffer with shellcode and then executing it, and address space layout randomization to prevent an attacker from easily finding library gadgets that may be assembled on the stack and run. But some stack manipulation is still possible even if all the protections are active. For example, consider the following code which is provided in file stk-ovrw-1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char *name[2];

void g() {  execve(name[0], name, 0x0); }

long *f (long *x) {
   *(&x+11) = (long*)g;
   return 0;
}

void h(int thevariable) {
   long *s = (long*)123;
   if (thevariable > 10) f(s);
}

int main (int argc, char **argv) {
   if (argc < 2) {
      printf("Usage: stk-ovwr-1 <positive-integer>\n");
      return 0;
   }
   name[0] = (char*)malloc(20);
   strcpy(name[0],"/bin/dash");
   name[1] = 0x0;
   h(atoi(argv[1]));
   return 0;
}
```

This code is compiled with all protections active. In function `main`, function `h` is called with an argument that is taken from the command line and converted to a positive number. The function `h` defines pointer `s` and calls function `f` with argument `s` if the command line argument is a number greater than 10. Since `s` is on the stack, its address can be used to access any point in the stack. This happens in function `f` and then it appears that `f` returns with value 0, which is ignored by `h` as it returns to `main` ending the execution without effect. However, in `f`, the value of `x` is used to get an address on the stack like this: `&x`. Then, `(&x+11)` provides the location of the return address back to the caller `h`. But this is overwritten with `*(&x+11) = (long*)g` where `(long*)g` is the address of function `g`.

Hence, instead of returning to `h` from `f`, execution proceeds to `g` which invokes a dash shell. Running the code with argument 42 results in $, the prompt of the dash shell. Type `$$` in the dash shell to get something like this: `/bin/dash: 1: 488502: not found` which reveals what has happened. Again, all protections are active.

Consider the following code which is provided in file mem-unsafe-4.c:

```
#include <stdio.h>

   int main() {
      int i;
      int x = 6;
      char y[] = "0123456789";
      char* z = x + y;
      printf(" z:%p  y:%p\n", z, y);
      printf("&z:%p &y:%p &x:%p\n", &z, &y, &x);
      printf("z[0..9]: ");
      for (i=0 ; i < 10 ; i++) printf("[%c]",(char)z[i]);
      printf("\ny[0..9]: ");
      for (i=0 ; i < 10 ; i++) printf("[%c]",(char)y[i]);
      printf("\n");
   }
```

This compiles and runs. But `z` points to a memory address six characters beyond `y`, and if z is in some way intended to print some of the contents of y then it will start six characters into the string y. But printing the last six of ten characters is memory that should not be allowed to be accessed. Output, compiled with gcc v.11.3.0, is this:

```
    z:0x7ffcb68353f3  y:0x7ffcb68353ed
   &z:0x7ffcb68353e0 &y:0x7ffcb68353ed &x:0x7ffcb68353d8
   z[0..9]: [6][7][8][9][][][3][Ã][Â][Â]
   y[0..9]: [0][1][2][3][4][5][6][7][8][9]
```

Observe that not even a segmentation fault occurs.

Consider the following code which is provided in the file `mem-unsafe-5.c`:

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

   int main (int argc, char **argv) {
      int x = 1;
      char buf [100];
      snprintf (buf, sizeof(buf), argv[1]);
      buf [sizeof(buf)-1] = 0;
      printf("Buffer size is: (%d) \nData input: %s \n", strlen(buf), buf);
      printf("x equals: %d (%#x)\nMemory address for x: (%p) \n",x,x,&x);
      return 0;
   }
```

Unfortunately, the format string can be modified at runtime by using specifiers in buf. Then, when run, information that should not be accessible is printed. Compile and run it with:

```
   ex10 "Hello %x %x"
```

Then the first `printf` line becomes

```
printf("Buffer size is: (23) \nData input: Hello %x %x \n");
```

so the output becomes:

```
Buffer size is: (23)
Data input: Hello 585b2da8 70c1af10
x equals: 1 (0x1)
Memory address for x: (0x7fff61baef0c)
```

try this:

```
mem-unsafe-5 "%s%s%s"
mem-unsafe-5 "%s%s%s%s"
mem-unsafe-5 "%s%s%s%s%s"
 ...
```

until a seg fault occurs.

A C/C++ function may request memory from the operating system. The memory is provided from the ‘heap’. The following code, provided in file `heap-ovrw-1.c`, shows a possible memory problem:

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

int main() {
   long diff, size = 8;
   char *buf1;
   char *buf2;

   buf1 = (char*)malloc(size);
   buf2 = (char*)malloc(size);

   if (buf1 == NULL || buf2 == NULL) {
      perror("malloc");
      exit(-1);
   }

   diff =  (unsigned long)buf2 - (unsigned long)buf1;
   printf("buf1 = %p, buf2 = %p, diff = %ld\n", buf1, buf2, diff);
   memset(buf2, '2', size);
   printf("BEFORE: buf2 = %s\n", buf2);
   memset(buf1, '1', diff+3);  /* We overwrite 3 chars */
   printf("AFTER:  buf2 = %s\n", buf2);

   return 0;
}
```

This function creates two 8 character buffers, pointed to by `buf1` and `buf2`, from the heap. It compiles with no warnings or errors and with all protections active. The operating system provides two non-overlapping chunks of memory of at least 8 characters in size. The variable `diff` holds the difference in ‘bases’ addresses, in bytes, of the chunks (when compiled and run `diff` is printed as 32). Buffer `buf2` is set to contain all characters ‘2’. This is shown by the first `printf` line. Then buffer `buf1` is set to contain all ‘1’ characters except that the number of ‘1’ characters assigned to buf_1 is ‘diff+3’ (which is 35) instead of size (which is 8). Since `buf2` points to an address `diff` higher than the address pointed to by `buf1`, the ‘1’ characters overwrite memory assigned to `buf1` and memory between the end of `buf1` and the beginning of `buf2` plus 3 extra characters which happen to be in `buf2`. All this happens without any error messages. The result is some of `buf2`'s characters are '1' instead of '2'. Here is the output from one run:

```
buf1 = 0x55e7cafdf2a0, buf2 = 0x55e7cafdf2c0, diff = 32
BEFORE: buf2 = 22222222
AFTER:  buf2 = 11122222
```

Cryptol is not memory safe but has a feature that checks whether a function is memory safe: for example it checks for potential array out-of-bounds errors. That feature is the invocation of `:safe <function>` at the Cryptol prompt. The following discussion illustrates how that works. Consider the Cryptol file `saf.cry` which contains the following two functions:

```
usaf: [1][8] -> [10][8] -> [1][8]
usaf a b = [(b@(a@0))]

saf : [1][8] -> [10][8] -> [1][8]
saf a b = if a@0 > 9 then [0]
          else if a@0 < 0 then [0]
          else [(b@(a@0))]
```

Both functions return the value of an element of array `b` that is indexed using the value of the single element of array `a`. But `usaf` makes no out-of-bounds checks on `b`, which must have 10 elements, and `saf` makes that check. Start Cryptol. Then, do what is shown in blue and observe the results:

```
version 2.12.0
https://cryptol.net  :? for help

Loading module Cryptol
Cryptol> :l saf.cry
Loading module Cryptol
Loading module Main

Main> :safe usaf
Counterexample
usaf
  [0x0a]
  [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
  ~> ERROR
invalid sequence index: 10
-- Backtrace --
(Cryptol::@) called at saf.cry:3:14--3:21
Main::usaf called at <interactive>:2:7--2:11
<interactive>::it called at <interactive>:2:7--2:11
(Total Elapsed Time: 0.014s, using "Z3")

Main> :safe saf
Safe
(Total Elapsed Time: 0.008s, using "Z3")
Main>
```

So, according to Cryptol, `usaf` is not safe (btw, safety goes way beyond memory safety as will be seen later) and a counterexample proving that is presented – all this counterexample does is try to index into element 10 which is beyond the dimension of the array (only indices 0 through 9 should be allowed). However, Cryptol declares function `saf` to be safe.

Next, the use of this feature is illustrated. Consider the C program below which is in file `saf.c`:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

uint8_t* saf (uint8_t *a, uint8_t b[10]) {
   if (*a < 0 || *a > 9) *a = 0;
   else *a = b[*a];
   return a;
}

int main (int argc, char **argv) {
   uint8_t array[10];
   uint8_t i, idx;

   for (i=2 ; i < 12 ; i++) array[i-2] = (uint8_t)atoi(argv[i]);
   idx = (uint8_t)atoi(argv[1]);
   uint8_t* out = saf(&idx, array);
   printf("%d\n", *out);
}
```

The C function `saf` does what the Cryptol function `saf` does. Compile `saf.c` to `saf` then run this:

```
[prompt]$ saf 4 0 1 2 3 4 5 6 7 8 9
```

The output is

```
4
```

The C function `usaf`, in file `usaf.c`, looks like this (`main` is the same and omitted):

```
uint8_t* usaf (uint8_t *a, uint8_t b[10]) {
   *a = b[*a];
   return a;
}
```

So, this function skips out-of-bounds checks on array `b`.

The Cryptol function `saf` plus `SAW` can be used to show that the C function `saf` is safe but not the C `usaf` function. First create llvm bitcode files from the C files like this:

```
clang -g -O0 -emit -llvm -c saf.c -o saf.bc
clang -g -O0 -emit -llvm -c usaf.c -o usaf.bc
```

The following is the contents of `saf.saw`:

```
import "saf.cry";

let safe_setup = do {
   arr <- llvm_fresh_var "array" (llvm_array 10 (llvm_int 8));
   parr <- llvm_alloc (llvm_array 10 (llvm_int 8));
   llvm_points_to parr (llvm_term arr);

   idx <- llvm_fresh_var "index" (llvm_array 1 (llvm_int 8));
   pidx <- llvm_alloc (llvm_array 1 (llvm_int 8));
   llvm_points_to pidx (llvm_term idx);

   llvm_execute_func [ pidx, parr ];
   llvm_points_to pidx (llvm_term {{ saf idx arr }});
};
```

```
let main : TopLevel () = do {
   m <- llvm_load_module "saf.bc";
   saf_proof <- llvm_verify m "saf" [] false safe_setup yices;
   print "Done!";
};
```

See Lesson 5 for how to build this SAW file. Run it like this:

```
saw saf.saw
```

The resulting output is this:

```
[21:57:21.360] Loading file ".../saf.saw"
[21:57:21.474] Verifying saf ...
[21:57:21.475] Simulating saf ...
[21:57:21.478] Checking proof obligations saf ...
[21:57:21.500] Proof succeeded! saf
[21:57:21.500] Done!
```

So, function `saf` in `saf.c` is safe (the particular concern here is memory safety but the result is that the function `saf` in `saf.c` is safe in other ways as well).

Now consider trying to verify C function `usaf` against Cryptol function `saf`. The llvm bitcode `usaf.bc` has already been generated above. Write `usaf.saw` as a copy of `saf.saw` except replace `saf.bc` with `usaf.bc` and `saf` with `usaf`. Run `saw usaf.saw` and get the following:

```
 Undefined behavior encountered
Details:
  Addition of an offset to a pointer resulted in a pointer to an address outside
  of the allocation

[11:11:41.728]SolverStats{solverStatsSolvers=fromList["SBV→Yices"],solve...}
[11:11:41.728] ----------Counterexample----------
[11:11:41.729]   index: [128]
[11:11:41.729] ----------------------------------
[11:11:41.729] Stack trace:
"llvm_verify" (/.../usaf.saw:20:18-20:29):
Proof failed.
```

So, the C `usaf` is not equivalent to the Cryptol `saf` and is not memory safe as the counterexample shows.