Buffer overflow vulnerabilities in CUDA: a preliminary analysis

Andrea Miele

EPFL, LACAL, Lausanne, Switzerland

arXiv:1506.08546v1 [cs.CR] 29 Jun 2015

Abstract

We present a preliminary study of buffer overflow vulnerabilities in CUDA software running on GPUs. We show how an attacker can overrun a buffer to corrupt sensitive data or steer the execution flow by overwriting function pointers, e.g., manipulating the virtual table of a C++ object. In view of a potential mass market diffusion of GPU accelerated software this may be a major concern.

1 Introduction

General-Purpose Computing on Graphics Processing Units (GPGPU) has become very popular in recent years. Modern GPUs are many-core accelerators for massively parallel applications rather than simply 2D and 3D graphics rendering coprocessors as in the past. Hardware/software platforms for GPGPU like Compute Unified Device Architecture (CUDA) [6] and OpenCL [3,1] allow developers to program graphics cards in languages like C/C++ or Fortran through APIs and language extensions. Such platforms have been used to accelerate a variety of scientific applications in the recent past and their use to speed-up commodity applications (e.g., web applications [4]) or operating systems can be expected in the near future [13]. As a consequence, questions about the security of code running on these platforms have become relevant. So far the security of GPU code has been somehow touched upon only considering GPUs as accelerators for malware [14,9] and more recently information leakage through side channels [11], but as far as we know no one has explored potential vulnerabilities of GPU code and their exploitation. A natural question that arises is whether or not classic C/C++ "buffer" overflow vulnerabilities [8,2,10] can be exploited. Leveraging a vulnerability to run arbitrary code on CPUs usually aims at interacting with the operating system via system calls to perform privilege escalation. In the context of GPUs there is no notion of operating system and the code runs on many cores across

a large number of threads. Thus, the concept of exploit assumes a different dimension. The ultimate goal of an attacker may be tampering with a parallel computation to maliciously affect the outcome or to force one or more threads to jump to specific parts of the code. This may become critical in view of future tight integration of CPU and GPU architectures.

The study of this problem on current GPUs is hindered by the lack of documentation on low level details of the hardware/software interfaces. The actual ISA of GPUs is usually not exposed by vendors who provide, in some cases, only virtual low level assembly [7]. Recently NVIDIA has released a disassembler for their binary executable format and a list of ISA instructions [5]. The latter, though, does not include a description of their syntax and functioning. Consequently, the analysis of GPU software vulnerabilities has to rely on "trial and error" experiments and reverse engineering.

In this paper we present a preliminary study of buffer overflow vulnerabilities in CUDA software. In particular we show how a buffer overrun can be exploited to overwrite function pointers (e.g., to manipulate the virtual table of a C++ object) to steer the execution flow. An attacker can hijack each call to specific functions to other functions in the code of his/her choice or perform limited return orient programming (ROP) [12]. We hope that our results will act as a wake-up call for the community as we believe that GPU software security will soon become an extremely relevant problem. The source code of this project will be made freely available.

2 Compute Unified Device Architecture (CUDA)

In this section we give a brief overview of CUDA. CUDA [6] is a computing platform, consisting in both a hardware and a software architecture, enabling NVIDA GPUs to support general purpose computing. At the programming level CUDA consists of extensions to the C/C++ language, libraries and some specific data types that enable the programmer to compute on the GPU. An actual function call mechanism exists and defining recursive functions is possible. Moreover, OOP programming is possible through a subset of C++ constructs that are supported. In CUDA programs the code that runs on the GPU is enclosed in a special function called *kernel*. A kernel is executed in the form of multiple parallel instances corresponding to a set of parallel *threads*. Threads are grouped in *blocks* and blocks are grouped in *grids*:

- **thread:** A thread executes one instance of the kernel, and it is uniquely identified inside its block by a thread identifier. Each thread has its program counter, registers, per-thread private memory, input, and output results.
- **block:** A block is a set of concurrently executing threads that can cooperate among themselves through synchronization and shared memory. Each thread block has a private per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms.
- **grid:** A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. The GPU executes a kernel as a grid of parallel thread blocks.

This hierarchal grouping scheme allows CUDA applications to scale across different device models. See [6] for more details.

3 Practical analysis

In this section we present our practical analysis through two examples. We will consider the presently common CPU/GPU interaction model in which the CPU and GPUs work on separate physical memory. The CPU loads the input data into the GPU memory and then runs the GPU kernel that will process these data and produce output data. A soon as the kernel terminates, the CPU copies the produced output from the GPU memory to the CPU memory. We will not explore the case where CPU and GPU are somehow integrated and share the same memory.

In our first example we show how function pointers in static memory can be overwritten to have each thread call a function that should normally not be callable. In our second example we show how the VTABLE of a C++ object in dynamic memory can be manipulated for the same purpose. Table 1 shows the specifications of the platform we have used for our experiments.

Table 1. Experimental platform specifications.

CPU	Intel Core i7-4790, 3.60GHz
GPU	Asus GTX Titan Black, 2880 cores
OS	Ubuntu 14.04.1 LTS 64-bit
Compiler	CUDA nvcc 6.5

3.1 A stack overflow

Consider the following fragment of CUDA code (the qualifier `__device__` denotes a function that is called by threads running on the GPU).

```
#define BUF_LEN 16

typedef unsigned long(*pFdummy)(void);

__device__ __noinline__ unsigned long dummy1(){
 return 0x1111111111111111;
}
__device__ __noinline__ unsigned long dummy2(){
 return 0x2222222222222222;
}
__device__ __noinline__ unsigned long dummy3(){
 return 0x3333333333333333;
}
__device__ __noinline__ unsigned long dummy4(){
 return 0x4444444444444444;
}
__device__ __noinline__ unsigned long dummy5(){
 return 0x5555555555555555;
}
__device__ __noinline__ unsigned long dummy6(){
 return 0x6666666666666666;
}
__device__ __noinline__ unsigned long dummy7(){
 return 0x7777777777777777;
}
__device__ __noinline__ unsigned long dummy8(){
 return 0x8888888888888888;
}
__device__ __noinline__ unsigned long dummy9(){
 printf("HELLO ADMIN!\n");
 return 0x9999999999999999;
}

__device__ __noinline__ unsigned long unsafe(
unsigned int * input, int len){

 unsigned int buf[BUF_LEN];
 pFdummy fp[8];
 fp[0]=dummy1;
 fp[1]=dummy2;
 fp[2]=dummy3;
 fp[3]=dummy4;
 fp[4]=dummy5;
 fp[5]=dummy6;
 fp[6]=dummy7;
```

```
 fp[7]=dummy8;
 unsigned int hash = 5381;

 // copy input to buf
 for(int i=0;i<len;i++)
  buf[i]=input[i];
 //djb2
 for(int i=0;i<BUF_LEN;i++){
  hash = ((hash << 5) + hash) + buf[i];
 }
 return (unsigned long) (fp[hash%8])();
}
```

The function `unsafe` is vulnerable to a buffer overflow (the array `buf` can be overridden if the attacker has control over the variables `string` and `len`). It computes a hash value of the input (using the djb2 algorithm of D.J. Bernstein) and uses such value reduced modulo 8 to select and call one of the first 8 “dummy” functions. The code of the simple CUDA kernel using the `unsafe` functions is the following:

```
__global__ void test_kernel(unsigned long* hashes,
unsigned int * input, int len, int *admin){
 unsigned long my_hash;
  int idx=blockDim.x*blockIdx.x+threadIdx.x;
 if(*admin)
  my_hash=dummy9();
 else
  my_hash=unsafe_hash(input+(len*idx),len);
 hashes[idx]=my_hash;
}
```

It is possible to use `cuda-gdb` or disassemble the binary with `cuobjdump` to figure out what are the addresses of the dummy functions. For instance on our platform the function `dummy9` has address `0x4e0`. This address is relative to the base address of the code section and does not change across multiple executions. We launch our kernel on one thread with value pointed by `admin` set to zero (we omit the very simple host code for the sake of clarity) and observe that if we fill the input buffer with at most 26 values (the value of `len` is set to 26), for instance 26 times `0x4e0`, the code prints correctly: `Hash[0]: 6666666666666666`. If we fill the input buffer with one more value `0x4e0` (the address of `dummy9`) and set the value of `len` to 27, the output is instead `HELLO ADMIN! Hash[0]: 9999999999999999`, thus we have successfully overwritten the function pointers with the address of `dummy9`. If more than one thread is run we observe that each thread is hijacked to execute `dummy9`. By looking at the disassembled binary with `cuobjdump`:

```
     ......................................
 /*03d8*/  LDL R0, [R0];
 /*03e0*/  PRET 0x3f0;
 /*03e8*/  BRX R0 -0x3f0;
```

```
/*03f0*/  IADD R1, R1, 0x80;
/*03f8*/  RET;

/*0408*/  MOV32I R4, 0x11111111;
/*0410*/  MOV32I R5, 0x11111111;
/*0418*/  RET;
/*0420*/  MOV32I R4, 0x22222222
/*0428*/  MOV32I R5, 0x22222222
/*0430*/  RET;
/*0438*/  MOV32I R4, 0x33333333

/*0448*/  MOV32I R5, 0x33333333
/*0450*/  RET;
/*0458*/  MOV32I R4, 0x44444444
/*0460*/  MOV32I R5, 0x44444444
/*0468*/  RET;
/*0470*/  MOV32I R4, 0x55555555
/*0478*/  MOV32I R5, 0x55555555

/*0488*/  RET;
/*0490*/  MOV32I R4, 0x66666666
/*0498*/  MOV32I R5, 0x66666666
/*04a0*/  RET;
/*04a8*/  MOV32I R4, 0x77777777
/*04b0*/  MOV32I R5, 0x77777777
/*04b8*/  RET;

/*04c8*/  MOV32I R4, 0x88888888
/*04d0*/  MOV32I R5, 0x88888888
/*04d8*/  RET;
/*04e0*/  MOV32I R4, 0x0;
/*04e8*/  MOV32I R5, 0x0;
/*04f0*/  MOV R7, RZ;
/*04f8*/  MOV R6, RZ;

/*0508*/  JCAL 0x0;
/*0510*/  MOV32I R4, 0x99999999
/*0518*/  MOV32I R5, 0x99999999
/*0520*/  RET;
/*0528*/  BRA 0x528;
/*0530*/  NOP;
/*0538*/  NOP;
    ......................................
```

we observe that the address `0x4e0` of `dummy9` is relative to the base address of the code as mentioned above. The “Branch to Relative Indexed Address” `BRX` instruction at address `0x03e8` is used to jump to the correct dummy function, whose address is stored in register `R0`.

It follows that by overwriting the function pointers in `fp` an attacker can jump to any address in the code memory and so ROP type of exploits are theoretically possible. Notice that the function call and return mechanism is handled with a `PRET` instruction followed eventually by a `RET` instruction. The `PRET` instruction presumably stores the return address (again relative to the code base address) at an unknown location. Classic return address overwrite attacks seem not to be feasible and attempts to jump outside the code memory failed (for instance we could not find a way to jump to shellcode injected into the buffer `buf`). Therefore, we conclude that code and data address spaces are separated and thus simple injected code execution is not possible.

3.2 A "heap overflow": manipulating the virtual table of a C++ object

Consider the following code where we define a C++ class B with four virtual methods: `f1`, `f2`, `f3` and `f4` and the derived class D defines the above four methods. The function `unsafe` is very similar to the homonymous function described in section 3.1, but in this case the array `buf` and a class D object are dynamically allocated. The function is vulnerable to a "heap" overflow as the array `buf` can be overridden to overwrite the adjacent class D object.

```
#define BUF_LEN 8

class B    //base class
{
  public:
    __device__  virtual unsigned long f1
    (unsigned int hash)
     {return 0;}
    __device__  virtual unsigned long f2
    (unsigned int hash)
     {return 0;}
    __device__  virtual unsigned long f3
    (unsigned int hash)
    {return 0;}
    __device__  virtual unsigned long f4
    (unsigned int hash)
    {return 0;}
};

class D : public B {
  public:
    __device__ __noinline__ unsigned long f1
    (unsigned int hash);
```

```
    __device__ __noinline__ unsigned long f2
    (unsigned int hash);
    __device__ __noinline__ unsigned long f3
    (unsigned int hash);
    __device__ __noinline__ unsigned long f4
    (unsigned int hash);
};
__device__ __noinline__ unsigned long D::f1
(unsigned int hash)
{return hash;}
__device__ __noinline__ unsigned long D::f2
(unsigned int hash)
{return 2*hash;}
__device__ __noinline__ unsigned long D::f3
(unsigned int hash)
{return 3*hash;}
__device__ __noinline__ unsigned long D::f4
(unsigned int hash)
{return 4*hash;}

__device__ __noinline__ unsigned long secret() {
  printf("HELLO ADMIN! ");
  return 0x9999999999999999;
};

__device__ __noinline__ unsigned long unsafe(
unsigned long * input, unsigned int len){
  unsigned long res =0;
  unsigned long hash = 5381;
  unsigned long *buf =
  (unsigned long *)
  malloc(sizeof(unsigned long)*BUF_LEN);
  D *objD = new D;

  // copy input to buf
  for(int i=0;i<len;i++)
    buf[i]=input[i];
  //djb2
  for(int i=0;i<BUF_LEN;i++){
    hash = ((hash << 5) + hash) +buf[i];
  }

  res=objD->f1(hash);
  res=objD->f2(res);
  res=objD->f3(res);
  res=objD->f4(res);
  free(buf);
  delete objD;
  return res;
```

```
}
```

The kernel we focus on is also very similar to the kernel presented in section 3.1:

```
__global__ void test_kernel(unsigned long* hashes,
unsigned long * input, unsigned int len, int *admin){
  unsigned long my_hash;
  int idx=blockDim.x*blockIdx.x+threadIdx.x;
  if(*admin)
    my_hash=secret();
  else
    my_hash=unsafe_hash(input+(len*(idx)),len);
  hashes[idx]=my_hash;
}
```

By instrumenting the code with simple `printf` calls we have been able to observe the following:

- The addresses of dynamically allocated memory blocks (`malloc`) or objects (`new`) are predictable. In our case, considering the first thread of the first block, the address of the array `buf` allocated in the function `unsafe` is always `0xb0513f920`.
- The first 64-bit value stored in an object of class D is the address of its virtual table (VTABLE). This address is the same across different threads as one would expect.
- The VTABLE contains the addresses (relative to the base of code section) of the four virtual functions defined in class D. This can be verified printing the VTABLE and comparing the first four 64-bit values with the addresses of the four functions appearing in the disassembled code obtained with `cuobjdump`.

Using the above observations we can exploit the overflow vulnerability to overwrite the VTABLE address of the class D object instantiated in a thread with the address of a "forged" VTABLE (which is stored in `buf`) containing 4 copies of the address of the function `secret`. Then the thread is forced to call the function `secret`.

Focusing on a single thread we have the memory layout showed in Figure 1. The exploit is achieved by filling the array `buf` as depicted in Figure 2, namely the first 11 locations of `buf` must contain the address of the the forged VTABLE, which is in turn stored in `buf` starting at the 12-th location. We have not discovered any hint suggesting the use of classic `malloc` linked lists with pointers stored next to the actual data. We leave the exploration of classic `malloc` and `free` exploits [2] as future work.

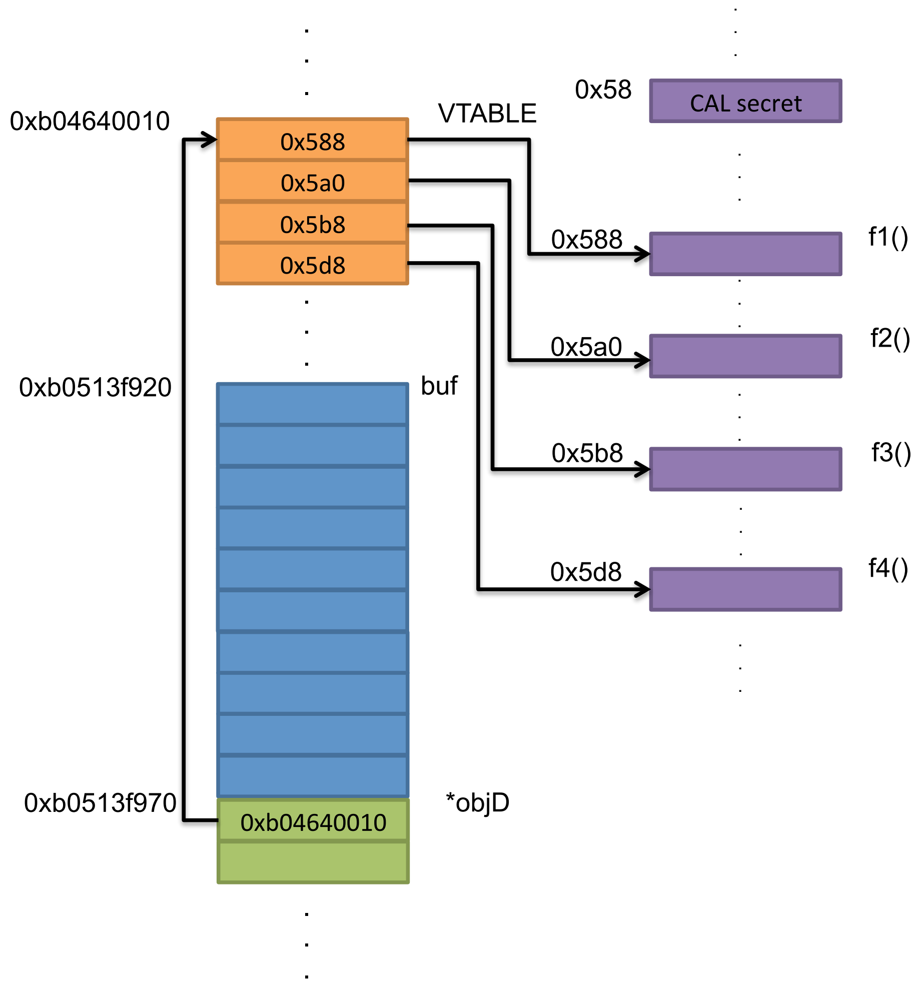

Fig. 1. Memory layout snapshot for function `unsafe`.

4 Conclusion

We have shown that CUDA software running on the latest NVIDIA GPUs can be vulnerable to classic buffer overflow attacks. Buffer overflows in both static and dynamic memory can be exploited to overwrite sensitive data or function pointers (e.g., to manipulate a C++ object virtual table). Our analysis does not expose any concrete threat, however it shows that the exploitation of vulnerabilities in GPU software is possible. This may

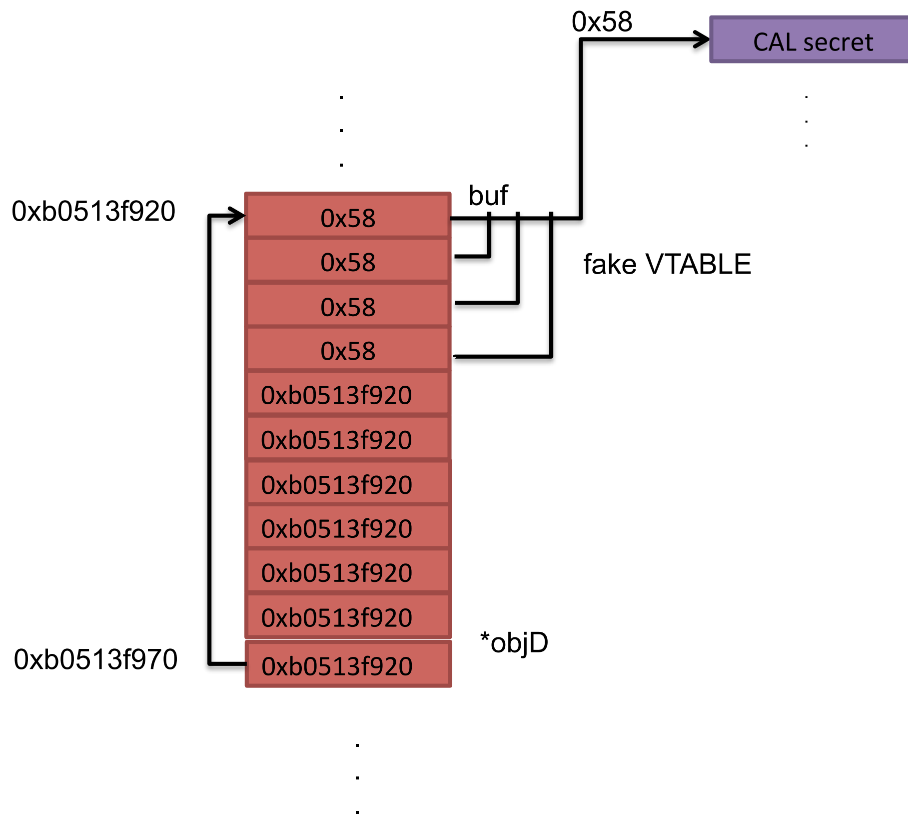

Fig. 2. Exploiting the heap overflow in function `unsafe`. We override the buffer `buf` to inject a forged VTABLE into the object `*objD`.

become a critical problem in the future if commodity GPGPU software will spread, especially if GPUs and CPUs will be tightly integrated.

References

1. AMD. http://developer.amd.com/tools-and-sdks/opencl-zone/.
2. anonymous. Once upon a free(). http://phrack.org/issues/57/9.html.
3. K. Group. OpenCL. `http://www.khronos.org/opencl/`.
4. K. Group. WebGL. `https://www.khronos.org/webgl`.
5. NVIDIA. CUDA binary utilities. `docs.nvidia.com/cuda/cuda-binary-utilities/index.html`.

6. NVIDIA. CUDA Programming Guide, docs.nvidia.com/cuda/cuda-c-programming-guide/. 2014.
7. NVIDIA. Parallel Thread Execution ISA docs.nvidia.com/cuda/parallel-thread-execution/. 2014.
8. A. One. Smashing the stack for fun and profit. http://phrack.org/issues/49/14.html.
9. D. Reynaud. GPU Powered Malware. Ruxcon 2008, 2008.
10. rix. SMASHING C++ VPTRS. http://phrack.org/issues/56/8.html.
11. A. V. Roberto Di Pietro, Flavio Lombardi. CUDA Leaks: Information Leakage in GPU Architectures. arXiv:1305.7383.
12. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security*, 15(1), Mar. 2012.
13. M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. *SIGARCH Comput. Archit. News*, 41(1):485–498, Mar. 2013.
14. G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Gpu-assisted malware. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 1–6, Oct 2010.