You Can Type, but You Can’t Hide

A Stealthy GPU-based Keylogger

EUROSEC 2013

Evangelos Ladakis

Lazaros Koromilas, Giorgos Vasiliadis, Sotiris Ioannidis, Michalis Polychronakis

(FORTH-ICS)
Outline

- Background
- A GPU-Based keylogger
- Evaluation
- Defenses
Keyloggers

- Malware that records keystrokes

Types:

Hardware (devices plugged in keyboard)

Software (user mode or kernel mode)

User mode:

They use OS functionalities:

- Character device files Linux OS
- GetAsyncKeyState Windows OS

Kernel mode:

They implement “Hook” functions

- Can be detected by AVs/anti-malware software
Motivation

- How can we hide the malicious code from AVs/anti-malware software?
- Is it possible to use the GPU for building a stealthier malware?
General-Purpose Programming on GPUs (GPGPU)

- GPUs can be programmed for general purpose computation
  - Familiar API as C language extensions
- Existing GPGPU frameworks
  - OpenCL (Universal Programming Language)
  - NVIDIA CUDA (For NVIDIA Graphics Cards)
- General-Purpose Programming is directly supported by most commodity drivers/video cards
  - A GPU-based keylogger will run without problems on most systems
Overall approach

- Scan kernel’s memory to locate the keyboard buffer
- Remap the memory page of the buffer to user space
- Set the GPU to periodically read and scan them for sensitive information (e.g., credit card numbers)
- Unmap the memory in order to leave no traces
Implementation

Step 1: Locate the keyboard buffer

- Keyboard buffer dynamically changes address after system rebooting or after unplugging and plugging back in the device
### Implementation

Scan the kernel memory using heuristics

<table>
<thead>
<tr>
<th>Struct URB (USB Request Block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct usb_device *dev</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>void *transfer_buffer</td>
</tr>
<tr>
<td>(actual keyboard buffer)</td>
</tr>
<tr>
<td>dma addr t *transfer_dma</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>u32 *transfer_buffer_length</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>struct usb_device</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>char* product</td>
</tr>
<tr>
<td>(describes the device)</td>
</tr>
</tbody>
</table>

*For proof of concept*
we scanned kernels memory with a kernel module

Must contains substrings

“USB”, “keyboard”
Implementation

Step 2: Configure the GPU to constantly monitor buffer contents for changes
Implementation

• The GPU driver allows DMA access **ONLY** to the host process' address space
  - *Only to memory regions allocated through a special CUDA API call*
• Use a kernel module to remap the physical page of the buffer to the user-level process' memory space
Implementation

Step 3: Start GPU process & Capture keystrokes
Implementation

• Uninstall the module

• Use polling to catch keystrokes
  ▪ “wake up” GPU process periodically through the CPU controller process

• Simple state machine translates keystrokes into ASCII characters

• Store keystrokes into Video RAM
Implementation

Step 4: **Scan captured keystrokes for sensitive information**

- GPU-based regular expression parser

<table>
<thead>
<tr>
<th>Credit card</th>
<th>Regular expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>VISA</td>
<td>^4[0-9]{12}(?:[0-9]{3})?$</td>
</tr>
<tr>
<td>MasterCard</td>
<td>^5[1-5][0-9]{14}$</td>
</tr>
<tr>
<td>American Express</td>
<td>^3[47][0-9]{13}$</td>
</tr>
<tr>
<td>Diners Club</td>
<td>^3(?:0[0-5]</td>
</tr>
<tr>
<td>Discover</td>
<td>^6(?:011</td>
</tr>
</tbody>
</table>
Evaluation

- Ubuntu Linux 12.10 with kernel v3.5.0
- Used CUDA 5.0 SDK
- Executable less than 4 KB

- Polling interval tradeoff:

  Monitoring granularity vs. CPU/GPU utilization

  - Low Frequency: *might miss keystroke events*
  - High frequency: *might cause detectable CPU/GPU utilization increase*
CPU Utilization

![Diagram showing CPU utilization over kernel invocation interval (msecs)].

- **Fastest Typists**
GPU Utilization

![Graph showing GPU Utilization vs. Kernel invocation interval (msecs)]
GPU Utilization

GPU utilization (percent) vs. Kernel invocation interval (msecs)

Fastest Typists
Possible Defenses

• Monitoring GPU access patterns
  ▪ Multiple/repeated DMAs from the GPU to system RAM

• Monitoring GPU usage
  ▪ Unexpected increased GPU usage
Current Prototype Limitations

• Requires a CPU process to control its execution
  ▪ Future GPGPU SDKs might allow us to drop the CPU controller process

• Requires administrative privileges
  ▪ For installing and using the module
  ▪ However the control process runs in user-space
    • No kernel injection needed or data structure manipulation, in order to hide
Conclusion

• GPUs offer new ways for robust and stealthy malware

• Presented a fully functional and stealthy GPU-based keylogger
  ▪ Low CPU and GPU usage
  ▪ No Device Hooking
  ▪ No traces left after exploitation
  ▪ User Mode application. No kernel injection needed
Thank you
Locate the keyboard buffer

#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))

for (i = 0; i < totalmem; i += 0x10) {
    struct urb *urbp = (struct urb *)__va(i);
    if ( ( (urbp->dev % 0x400) == 0) &&
         ((urbp->transfer_dma % 0x20) == 0) &&
         (urbp->transfer_buffer_length == 8) &&
         (urbp->transfer_buffer != NULL) &&
         strcmp(urbp->dev->product, "usb", 32) &&
         strcmp(urbp->dev->product, "keyboard", 32)) {

        /* potential match */
    }
}
Related Work

- DMA Malware “DAGGER” by: Patrick Stewin and Iurii Bystrovx
  - Implemented in Intel's Manageability Engine (it is used for remote Bios operations)

- GPU assisted malware by: Giorgos Vasiliadis, Michalis Polychronakis and Sotiris Ioannidis
  - GPU-based self-unpacking malware