Process Behavior Analysis with Focused Clustering: Milestone 1

Isaac Amann

2/25/2024

Contents

1 Tools

1.1 Spring Boot

The server acting as a back-end for the eventual web page and as the collection server for data points is being implemented in Spring Boot with Java. I went with this choice since I have previous experience with it and I plan on using a particular feature of Java to invoke the program that runs the C++ implementation of Dr. Che's Sow and Grow algorithm once I begin implementing the data analysis feature.

1.2 React

I intend to implement the front-end dashboard in React since I have previous experience using it and it suits the problem well. React makes it relatively simple to alter web pages based on data received from API calls.

1.3 HTTP / REST API

I am implementing a REST API to serve requests from data collection clients and from the web-based dashboard.

1.4 LaTeX

All project documentation is being written in LaTeX. The default article document class is being used along with a few basic packages for graphics and tables. I have also written some macros for use case tables and requirements tables.

2 Techniques

2.1 DLL Injection

The process monitor uses DLL injection in order to run code within a monitored process's address space. This is necessary since Windows does not provide a direct way of monitoring Windows API usage, so processes need to be modified at run time. I used one of the most common techniques for injecting my DLL. It involves using opening a thread in the target processes address space and then using LoadLibrary to load the DLL [2]. This can be seen in the following code segment:

```
//Load dll into process
void TrackedProcess::attach()
{
  HANDLE threadHandle;
  void* injectedLibAddress;
  HMODULE hkernel32 = GetModuleHandleA("Kernel32");

  //Allocate memory in target process
  injectedLibAddress = VirtualAllocEx(this->processHandle, NULL,
      sizeof(libPath), MEM_COMMIT, PAGE_READWRITE);

  //Write DLL path name to memory
  WriteProcessMemory(this->processHandle, injectedLibAddress, (void
      *)libPath, sizeof(libPath), NULL);

  //Create new thread and load dll into process
  threadHandle = CreateRemoteThread(this->processHandle, NULL, 0, (
      LPTHREAD_START_ROUTINE)GetProcAddress(hkernel32, "LoadLibraryA
      "), injectedLibAddress, 0, NULL);
}
```

2.2 Function Hooking

Since the Windows API does not provide a method for creating an event for API calls or other means of monitoring calls, the monitored Windows API calls need to be directly modified in the target process's address space to modify its behavior to include updating a count of the number of times it has been called.

This is achieved with a hook and trampoline function. The first 5 bytes of a target function are stored in a buffer and replaced with a 1 byte machine code JMP instruction followed by a four byte integer representing the distance from the target function to the hook function. A trampoline function would also be created that would contain the first five bytes that were replaced followed by another JMP instruction. Another 4 byte integer would follow this JMP instruction representing the address of the target function + 5 bytes to avoid running the altered code again. The hook function would then run any extra desired code or modify arguments before calling the created trampoline function to return control to the original target function[1].

I initially tried this method, but quickly ran into issues when attempting to implement the trampoline function. The first JMP instruction would consistently work since the distance between the addresses of the target function and hook would be small, but the memory allocated to contain instructions for the trampoline function would be further away in the address space and could not be reached with a 4 byte jump. This method without modifications would only work on 32 bit systems since it relies on the address size being 4 bytes to match the 4 byte input that a JMP instruction expects. There are some articles that describe some solutions for this including loading the 8 byte address into a register and using that as the input to JMP instead, but I decided to use Microsoft Detours for function hooking. An example of the process of hooking a function with Detours can be seen in the following code segment:

```
//Create a function pointer to the original API function
LPVOID(WINAPI* origVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize,
    DWORD flAllocationType, DWORD flProtect) = VirtualAlloc;

DetourTransactionBegin();
DetourAttach(&origVirtualAlloc, hookVirtualAlloc);
DetourTransactionCommit();

//Create a hook function that matches the parameters and calling
    convention (WINAPI or __stdcall) exactly
LPVOID WINAPI hookVirtualAlloc(LPVOID lpAddress, SIZE_T dwSize,
    DWORD flAllocationType, DWORD flProtect)
{
     //Modify parameters and add any additional code here
     if (PRINT_CALLS)
          std::cout << "Called VirtualAlloc\n";
     counterMap.at("VirtualAlloc")->incrementCall();
     //Call the original unmodified function using the function
          pointer created earlier
     return origVirtualAlloc(lpAddress, dwSize, flAllocationType,
          flProtect);
}
```

2.3 Inter-process Communication

Since monitored processes keep counts of calls to API functions in their address spaces, this data needs to be sent to the process monitor. Initially I went with a message passing model using a named pipe. The monitor would create a named pipe for each monitored process and the monitored process would just open it like a file and write messages to it. This approach works, but it relies on a fixed buffer in the monitor and a file write call to the pipe can block if the buffer becomes full. This would cause a program to either freeze or fail since it may not have been designed to be blocked after calling a monitored Windows API call. I switched to a shared memory model and this resolved several issues I was having with my program causing system instability. How shared memory was implemented in the system can be seen in the code segment below:

```
  // ******** DLL main that runs inside target process ******** //
    //Open shared memory from monitor process

    //Used a base name of Local/APIMonitor
    //The path for each process is this base name with the PID appended
    std::string pipeName = pipeBaseName;
    pipeName.append(std::to_string(GetCurrentProcessId()));
    std::wstring temp = std::wstring(pipeName.begin(), pipeName.end());
    LPCWSTR fullString = temp.c_str();
    //Open a handle to the file mapping created by the monitor
    sharedMemoryHandle = OpenFileMappingW(FILE_MAP_ALL_ACCESS, FALSE,
        fullString);
    //Catch the case where the handle is null, would cause a crash on
        attached process
    if (sharedMemoryHandle == NULL)
    {
        return true;
    }
    //Use MapViewOfFile to map the file to a pointer so it can be
        accessed like a normal memory location. Casting the pointer to a
         C struct that contains integers for counts for all monitored
        API calls
    callCountContainer = (CallCountContainer*)MapViewOfFile(
        sharedMemoryHandle, FILE_MAP_ALL_ACCESS, 0, 0, sizeof(
        CallCountContainer));
  // ******** Called in TrackedProcess constructor******** //
    //Create shared memory to pass counts from tracked process
    std::string pipeName = pipeBaseName;
   pipeName.append(std::to_string(PID));
   std::cout << pipeName << std::endl;
   pipeHandle = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,
      PAGE_READWRITE, 0, sizeof(CallCountContainer), pipeName.c_str());
   if (pipeHandle == NULL)
   {
    _tprintf(TEXT("Could not create file mapping (%d)\n"), GetLastError
        ());
   }
    //Map to pointer
   callCountContainer = (CallCountContainer*)MapViewOfFile(pipeHandle,
      FILE_MAP_ALL_ACCESS, 0, 0, sizeof(CallCountContainer));
```

3 Progress

The monitor program is mostly complete and work on the back-end/collection server has been started. Some basic user authentication and data collection API calls have been implemented on the back-end server. The client is also able to post data points after a process has exited.

In addition to work on my project, I have worked on a couple utility programs for work on the Sow and Grow algorithm as well as started exploring solutions for load balancing between threads. These two utility programs include a cluster visualizer script in Python and a Bitmap to CSV utility written in C so testing data sets can be created from the pixel data of images.

3.1 Challenges

3.1.1 System Instability

Many programs would noticeably slow down or break after being injected with an earlier version of my DLL. This was likely due to the function for writing to the named pipe used for inter-process communication either being slow or blocking if the buffer happened to fill up. After switching to shared memory, this does not appear to happen anymore. I have also added a check to exclude specific executable names or PID's from being monitored. In addition to explorer.exe, I will likely add most user-level Windows services to the exclusion list to reduce the risk of my system causing instability that would affect the end user's experience.

3.2 Work Log

3.2.1 1/25/2024

Created a separate thread in attached processes to periodically send messages to monitor instead of making a WriteFile call within the hooked function to reduce the risk of blocking a process.

3.2.2 1/26/2024

Wrote function hooks for wsock32 API functions.

3.2.3 1/27/2024

Started writing function hooks for Advapi32 API functions.

3.2.4 1/29/2024

Wrote more function hooks.

3.2.5 2/1/2024

Started hooking User32 API calls.

3.2.6 2/11/2024

Finished writing function hooks for all API calls monitored by the system.

3.2.7 2/14/2024

Switched from using a named pipe to shared memory for inter-process communication. Updated constructor for apiCallCounter to work with this change.

3.2.8 2/17/2024

Created prompt on monitor program to get UUID and API key from user. The credentials are stored on the Windows Registry and the user is not prompted again if they already exist.

3.2.9 2/18/2024

Created README file for project main page

4 Preliminary Results

The collection client is reliably able to attach to running processes and produce counts of calls to monitored Windows API functions. These counts are then sent to the server application by HTTP successfully. I had concerns that Windows API functions that were not invoked directly, but instead used within a function from another DLL or library would not be affected by the function hook. However, I noticed that my test program showed calls to the Windows API function VirtualProtect after using malloc to allocate memory even though the test program did not call VirtualProtect directly.

The counts produced by the system also seem to be correct when monitoring the test program. The test program makes a known number of Windows API calls so that the values produced by the monitor program can be compared. It will be difficult to verify results outside of programs written specifically for testing. Most software is closed source and behaves differently based on input and other factors changing the number of times they would call specific Windows API calls.

5 Meetings

5.1 1/31/2024

- Demonstration of data visualization tool
- Compared test results between DBSCAN and Sow and Grow

5.2 2/7/2024

- Discussed poor thread use from threads running out of work early and exiting before other threads

5.3 2/14/2024

- Demonstration of bitmap to csv utility for creating datasets that would exaggerate the load balancing problem for testing
- Discussed possible memory leak with algorithm when using large number of seed points with multiple threads

5.4 2/23/2024

- Discussed testing results for memory problem. Memory use problem not caused by a memory leak
- Discussed pseudo code for our approach to improving load balancing between threads for the algorithm

6 References

[1] Jayson Hurst. *Basic Windows API Hooking.* https://medium.com/geekculture/basic-windows-api-hooking-acb8d275e9b8. 2021.

[2] Red Team Notes. *DLL Injection.* https://www.ired.team/offensive-security/code-injection-process-injection/dll-injection. 2018.