Non-invasive (in-memory) instrumentation

Toshihito Kikuchi

Jul-30-2018

Agenda

- Background
- What it can do
- How it works
- How to use
- Demo 1: Bottleneck analysis of Chrome function
- Demo 2: Comparative analysis between BlinkGC and MemGC

- The latest slides are available in https://github.com/msmania/procjack/blob/master/clove/Intro.pdf

Background

- Many situations where you want to instrument a code
  - To measure the performance of some operations
  - To reproduce a race condition
  - To collect more logs to understand the behavior
  - etc.
- Instrumentation is not always easy
  - Time consuming
    (e.g. Chromium needs 5-6 hours to build unless you're a Googler or rich..)
  - Impossible if you don't have code or build environment
    (e.g. Customer's code, 3rd-party)
  - Compiler optimization varies
    (e.g. PGO build)

What it can do

- Non-invasive instrumentation enables you to
  - Inject your code in arbitrary places
    (including the middle of a function)
  - Without modifying the target program

- Based on two techniques
  - Reflective DLL Injection: to inject DLL into a running process
  - Microsoft Detours: to hook the existing code

- Some limitations
  - No support for IA64/ARM/Linux/Kernel-Mode
  - Some spots cannot be hooked due to the nature of Detours
  - You need to write some assembly code
    (Obviously this is fun, but there may be people who don't like it.)

How it works: Reflective DLL Injection

- Famous way to inject your code into a running process
  - Leveraging VirtualAllocEx and CreateRemoteThread, you can invoke your DLL's DLLMain in a target process
  - Used in Metasploit

- Advanced version
  - You can invoke a DLL-exported function instead of DLLMain
  - You can invoke a DLL-exported function with a string parameter that is passed into the function.

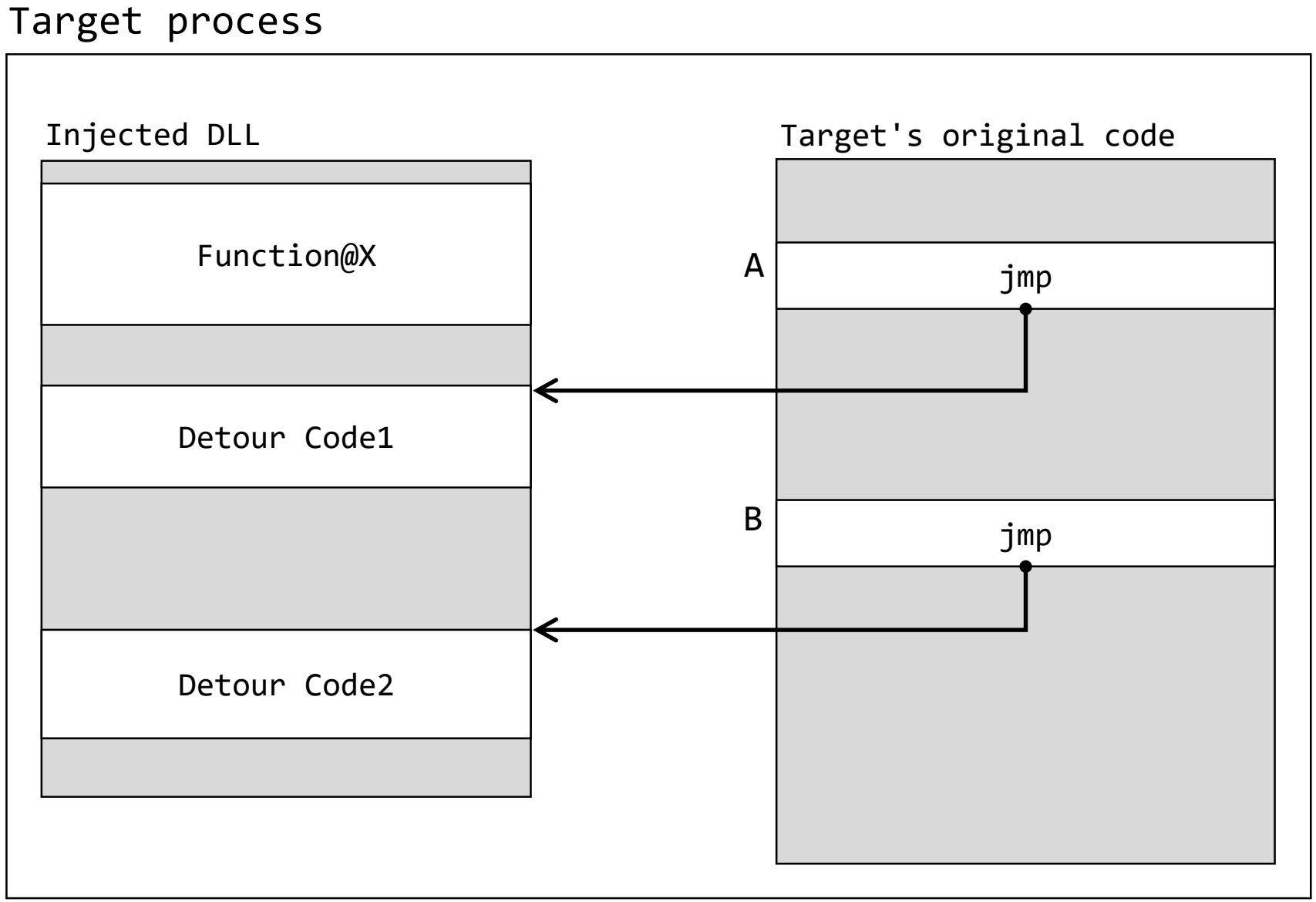
How it works: Microsoft Detours

- MSR's weapon to hook the code
  - Dynamically modifies the code to hook in an elegant way
    - 1. Disassembly the original code
    - 2. Move the original code to a different place as a trampoline function
    - 3. Put *jmp +rel32* on the hooking position
- Version 3.0 was available for a long time
  - Free version (Detours Express) supported only x86
  - Detours Professional supported x64, but it was $9,999.95
- Version 4.0.1 is now available
  - Open-sourced on GitHub
  - MIT License
  - Supports x86/x64/IA64/ARM/ARM64

How it works: DLL injection meets Detours

Injector command

Run Function@X which does:
- hooks A to jump Code1
- hooks B to jump Code2



How it works: Detouring details

- Detours basically detours a function's start address

Before Detouring

```
TargetFunction:
  sub   rsp,28h
  mov   r9,r8
  mov   r8,rdx
  mov   rdx,rcx
  ...
  ret
```

After Detouring

```
TargetFunction:
  jmp   DetourFunction
  int   3
  int   3
  mov   r8,rdx
  mov   rdx,rcx
  ...
```

```
DetourFunction:
  <your pre code>
  call TrampolineFunction
  <your post code>
  ret
```

```
TrampolineFunction:
  sub   rsp,28h
  mov   r9,r8
  jmp   TargetFunction+7
```

Explanation in Detour's Wiki:
https://github.com/microsoft/detours/wiki/OverviewInterception

How it works: Detouring details

- A way to hook a code in the middle of a function

Before Detouring

```
TargetFunction:
  ...
  lea   edi,[r14+28h]
  cmp   word[rcx],r8w
  jbe   $+0Ch  (2ab7eed0)
  mov   rax,qword[rcx+8]
  ...
```

After Detouring

```
TargetFunction:
  ...
  lea   edi,[r14+28h]
HookPosition:
  jmp   DetourFunction
  int   3
  mov   rax,qword[rcx+8]
  ...
```

```
DetourFunction:
  <your code>
  jmp TrampolineFunction
```

```
TrampolineFunction:
  cmp   word[rcx],r8w
  jbe   $+4001EDA6h  (2ab7eed0)
  jmp   HookPosition+6
```

How to use

- Run a command like this:
  > pj.exe [-d] [-w] <PID> <FILE>[?ORDINAL] [ARGS]
- Clove.dll is an example DLL
  - Ordinal#1 to show ProcessMitigation status
  - Ordinal#2 to release all hooks
  - Ordinal#3 to print results on the debugger console
  - Ordinal#100 (Explained in Demo1)
    > pj.exe <PID> clove.dll?100 <Addr1>-<Addr2>-..-<AddrN>
    to measure CPU cycles of each range [AddrX - AddrX+1]
  - Ordinal#200 (Explained in Demo2)
    > pj.exe <PID> clove.dll?200 <AddrX>
    to trace function calls of a function starting from AddrX

Demo1

- Mission: Find a bottleneck of Chrome's layout code

  More specifically, where is the slowest operation in chrome_child!blink::Document::UpdateStyleAndLayoutTree?

- Plan:
  - 1. Split the function into some ranges
  - 2. Measure CPU cycles of each range

- This demonstrates:
  - Hook the code in the middle of a function
  - Multiple injected codes interact with each other

Demo2

- Mission:
  Find the heap allocation pattern of BlinkGC and MemGC
- Plan:
  - Trace function calls of the following functions:
    - chrome_child!blink::Node::AllocateObject
    - edgehtml!MemoryProtection::HeapAllocClear<1>
  - Collect the following information
    - Caller's TID
    - Size to allocate
    - Return address
    - CPU cycles of each function call
- This demonstrates:
  - Invoke a C++ function from the hook
  - Handle both stdcall and cdecl in x86

References

- GitHub repo
  https://github.com/msmania/procjack/
- Microsoft Detours
  https://www.microsoft.com/en-us/research/project/detours/
  https://github.com/microsoft/detours