Karl Lewis
Real-Time Debugging

Project 3: Real-Time Bug Tools Report

# Scenario 1: Unity Plugin Inaccessible Back-End Code

## Description of Issue/Scenario

The core issue with these types of bugs is related to problems within the core framework/backend code used to drive a plugin where the front end developer would not have access to the plugin's source code, effectively **preventing the bugs from being fixed** without special tools or the source code itself.

## Details of Issue/Scenario

These issues can pertain to a variety of different bugs within the plugin's backend code, but the main problem is the fact that this code remains **inaccessible** to front end developers. Without access to the code of the plugin itself, a developer cannot feasibly be able to fix the bugs with the plugin in the context of their application. These issues can be even further compounded depending on the context of the application being built with the plugin; perhaps the issue stems from the plugin's interaction with the front-end code, for example. In any case, without **access to the source code of the plugin** or some way to **reverse engineer** it source code, bugs in the back-end will be difficult if not impossible to solve by a front end developer.

What should be happening here is that the front end developer has **access to the code of the plugin** or some way of examining what's happening within it in order to squash the natively inaccessible bugs. Once access to the plugin's backend is accessible in any way, the developer will be able to proceed with debugging as normal, although they might need to retool their processes to work according to the specific tool in use.

## Discovery/Replication of Issue

A whole host of issues with inaccessible backend code can be either easy or incredibly difficult to detect depending on what's actually wrong in the backend. Issues can range from fatal crash bugs that immediately break the entire application before it is even compiled, run time errors that can occur

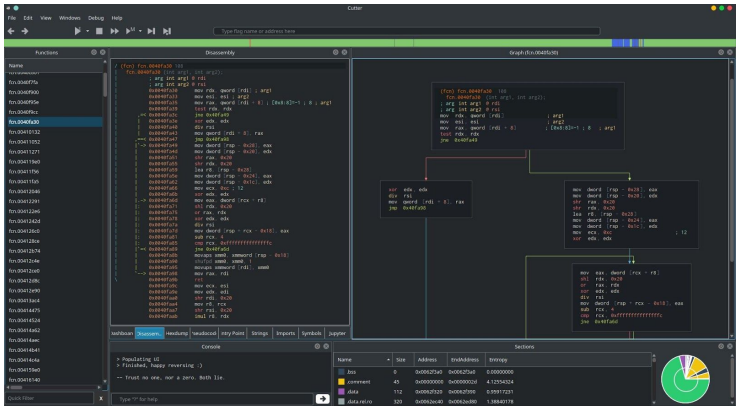
either frequently or randomly, or even subtle errors that only happen specifically to the context of the front end application.
The varying range of bugs that can occur in the backend makes the replication process vary from bug to bug as well. Some bugs will be replicated immediately when either opening the application or starting it for testing (fatal crashes, etc), others will need repeated running of certain processes or patterns to trigger successfully.

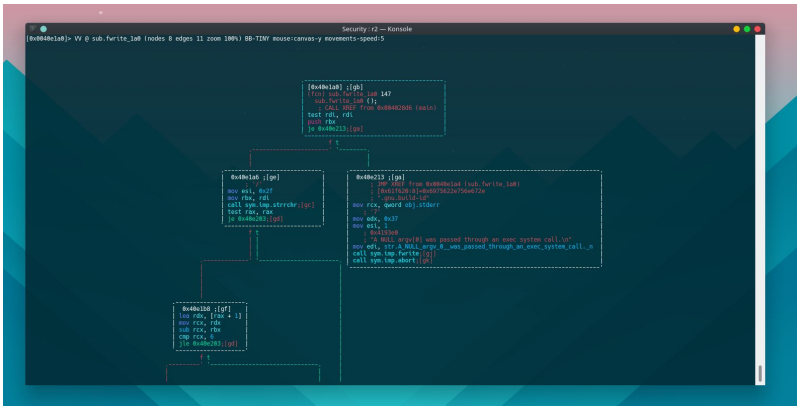
# Tool Research/Strengths and Weaknesses

Since these types of issues are related to code accessibility, a debugger would need to use a **reverse engineering** debugging tool in order to access and analyze the code. In the case of a Unity plugin, a reverse engineering tool would need to work with .dll files and be able to analyze the code within. Preferably, the tool should have an easy to use interface or commands depending on if it is a command-line tool or not.

Based on my research, I've identified the following possible tools that would work to solve these sorts of issues:

## Radare 2:





A reverse engineering debugging tool that works with binary files. It can analyze binaries, disassemble code, and features debugging programs that can be used for a variety of different debugging contexts

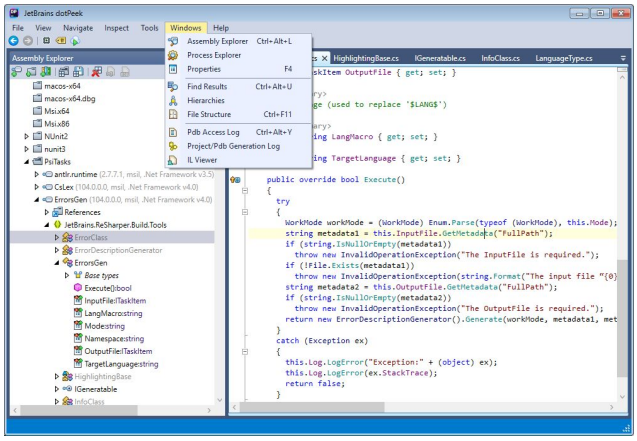
Strengths:

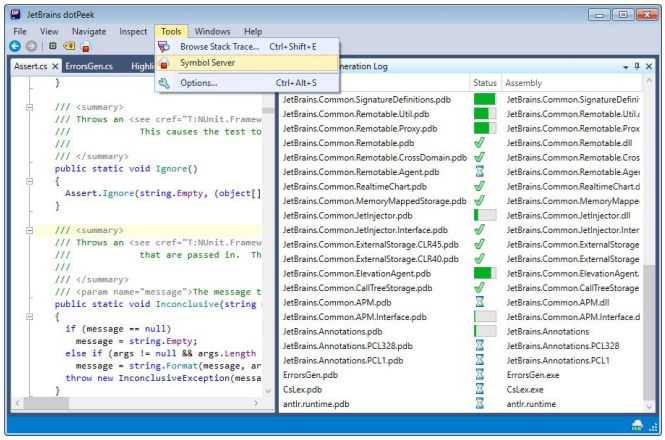
- Can debug without source code using disassembler
- Can debug using local or native debuggers
- Node-based command-line interface
- Supports virtually all platforms
- Free and open source
- Lots of documentation

Weaknesses:

- Originally intended for analysis instead of pure debugging; a little obtuse to get up and running to a debugging context
- Can require knowledge of assembly code in many instances
- Steep learning curve
- GUI are separate downloads; natively uses a command-line interface

## JetBrains dotPeek Decompiler





A standalone .NET assembly decompiler that decompiles .NET assemblies into C# code and works with the Visual Studio debugger

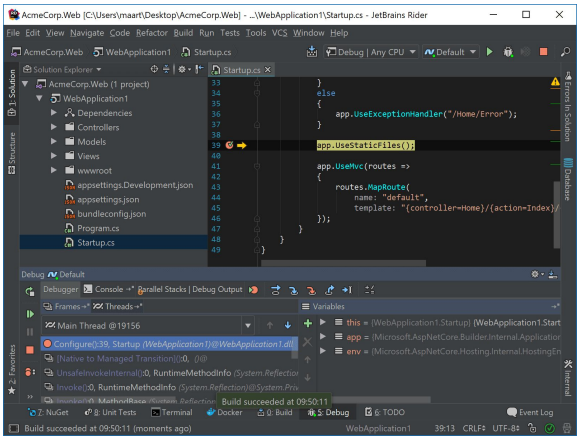
Strengths:

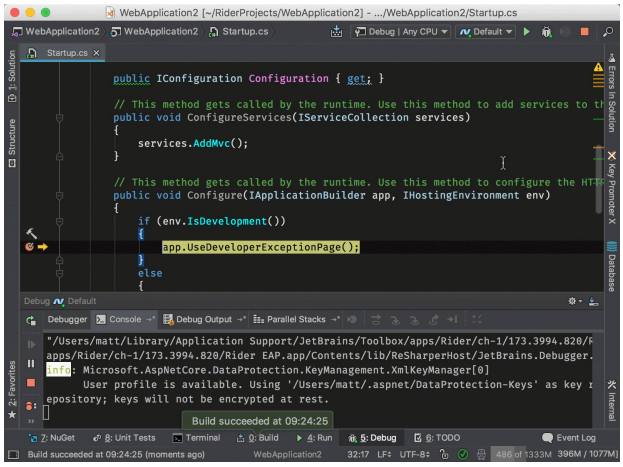
- Supports multiple formats, including .dll libraries, .exe executables, and .winmd Windows metadata.
- Decompiled code can be saved as a Visual Studio project and read and edited as C# code; useful for debugging third-party code quickly
- Can to provide the VS debugger with symbols needed to debug the assembly code; allows easy debugging of code even without the source
- Robust navigation and search tools
- Free standalone tool

Weaknesses:

- Not inherently a debugger on its own; merely decompiles assembly and requires another debugging tool (usually VS debugger)
- Requires PDB files for debugging; if no PDB files are present, files must be decompiled first by the tool and symbols must be manually set up in Visual Studio
- Complex to use

## [JetBrains Rider]





A feature-rich IDE similar to Visual Studio for cross-platform .NET projects. Features live code analysis, editing, refactoring, debugging, and unit testing tools among many other features.

Strengths:

- Excellent debugging tools that can be used to debug inaccessible third-party code in real-time without decompiling; effortless debugging of inaccessible source code.
- Breakpoints can be set in third party code with ease
- Debugging can be attached to a running process in real-time
- Full IDE suite; everything is integrated in one package
- Easy to use and highly robust feature set apart from just debugging

Weaknesses:

- Paid software
- Requires learning an entirely new development environment
- Would require a switch for platform development tool to the new IDE

# Solution Proposal

Based on the findings from the tool research process, I propose the following solutions depending on context:

### Small Scale Project/Studio/Student

Use either Radare 2 or dotPeek decompiler. These programs are relatively lightweight and work together with Visual Studio to trace through back-end code. The open-source nature of Radare 2 would be more optimal for more experienced developers that need to modify the debugging

environment if issues with the plugin are severe. dotPeek would be optimal for students or less experienced developers; while the process of getting .PDF files from the decompiled libraries is a little involved, it is relatively simple and can get the same access to third-party code that Rider provides for free. Either program would be useful for getting to the root of a plugin bug with inaccessible source code on the plugin end, with Radare 2 being more on the advanced side if issues are incredibly serious/fatal crashes whereas dotPeek would be for more run-time errors or less serious bugs.

### Large Project/Studio/AAA company

In the case where money is not an issue (aka at a large company with the funds to support a widespread company IDE change), JetBrains Rider is the easy winner for debugging inaccessible plugin code. The speed and ease with which third party code can be debugged is second to none for these sorts of issues, as the front-end developer could simply step into the inaccessible source code, which decompiles and injects it into the program in real-time. If the plugin in question is a particularly bug-heavy bastard, Rider would make debugging it a piece of cake. The only cons in this situation are obviously the price and the fact that the front-end developers would need to learn an entirely new IDE if they're used to Visual Studio, but the results might be worth it in the end with this level of debugging fidelity.

# Scenario 2: Networked Application Timeout Sabotaging Debugging Session

## Description of Issue/Scenario

The core issue with networked application bugs in this scenario is that using standard debugging methods like breakpoints in Visual Studio **causes one of the application's network loop to pause, causing timeout and compromising the debug session**.

## Details of Issue/Scenario

The root of these sorts of issues is independent of the actual bugs with the code itself; the bugs in the code can be literally anything. With this in mind, the problem is not with the bugs themselves, but with standard debugging tools like breakpoints in Visual Studio. When a breakpoint is triggered during a debugging session, the application pauses. This is fine for single client applications, but when two or more clients are connected to each other remotely, **the pausing occuring during debugging effectively stops the code from updating, sending, and receiving network packets,** effectively making the debugging process of network related issues impossible with standard debugging tools.

What should be happening here is that the debugger should be able to either seamlessly debug a networked application without pausing the execution of an update loop or use an entirely different method of debugging entirely for networked issues. Once application pausing is out of the equation, networked code can be debugged as normal to find any issues with the code without risk of a compromised debugging session due to timeouts and disconnections.

## Discovery/Replication of Issue

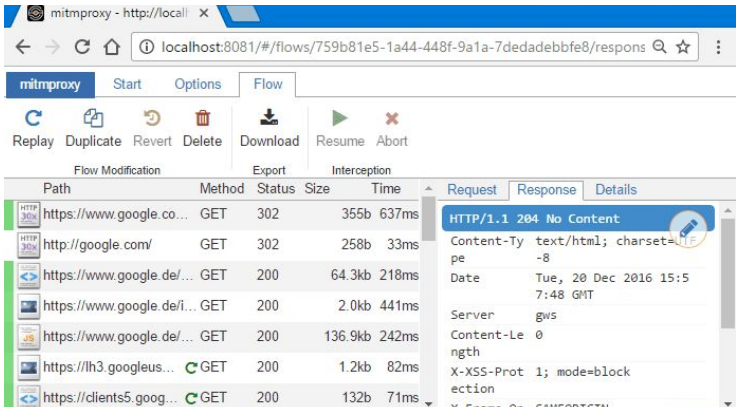
Issues with the source code of networked applications are not the cause of concern here, but rather the debugger itself stopping networked code from executing. Because of this, replicating the issue essentially requires two instances of the application to be connected to one another during a debugging session. When a breakpoint triggers on one of the application instances, the code execution (and consequently the network update loop) will stop executing. If the code execution is not resumed shortly after starting the debugging session with breakpoints, the triggered client will eventually time out from the other connection, causing the debug session to terminate prematurely.

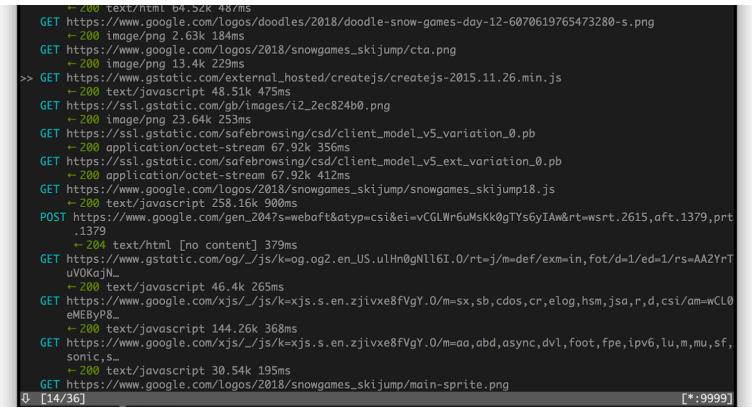
## Tool Research/Strengths and Weaknesses

Since this issue is related not to the code of an application itself but rather the debugging process used, a debugger would need to make use of **networking specific debugging tools** that **analyze the actual network packet data** while the program is running. With these tools, debuggers can see the same data being sent and received between two or more application instances without having to pause the code execution and cause an unwanted code termination.

Based on my research, I've identified the following two tools that are best suited for this sort of issue.

### [mitmproxy]




An interactive HTTPS proxy debugger that measures and decodes networked data. It can intercept, inspect, and modify networked data and is mainly tailored for web based connections.

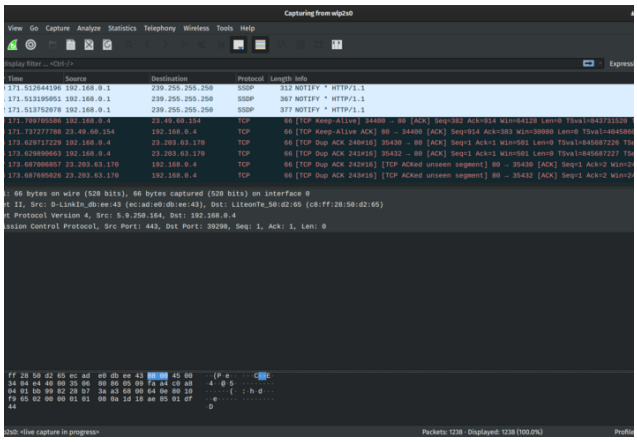
Strengths:

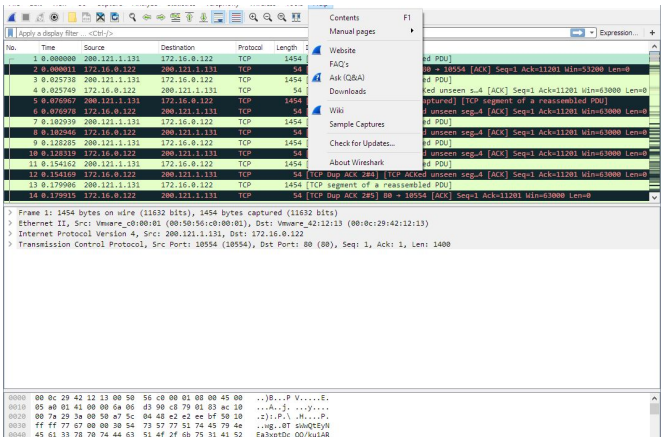
- Works with any web protocols
- Any type of networked message can be intercepted, analyze, or modified without interrupting code execution
- Features a GUI web based interface or a standard command line
- Free and open source
- Lots of documentation

Weaknesses:

- Mainly tailored for web application development
- Requires knowledge of SSL/TLS protocols
- Steep learning curve

## [Wireshark]





A network protocol analyzer that shows what's happening over the network at the lowest possible levels. Captures and analyzes data sent over the network in a variety of ways

Strengths:

- Allows deep inspection of virtually any protocol
- Live capture of network activity and packets prevents code pausing entirely
- Packet data is filtered through a robust GUI with many different filters
- Free and open source
- Widely used in various industries; tried and tested program
- Extensive and detailed documentation

Weaknesses:

- More of an analysis tool than debugging; however the analysis can *lead* to successful debugging
- Huge learning curve; lots of complex features

## Solution Proposal

Based on the findings from the tool research process, I propose the following solution.

It was rather difficult to find a lot of tools suited to this sort of situation, but the two that I did find are both used for analyzing network traffic and packet data. Using either of these tools for this purpose would effectively bypass the issue of debugging stopping the networking loop entirely; any and all data sent and received by the network (i.e, any data that the debugger would actually need running when connected to another remote) can be analyzed and monitored with these programs.

With this in mind, I propose that **Wireshark** is much better suited to this particular task than mitmproxy. Whereas mitmproxy is more tailored for web based development, Wireshark works with virtually any sort of networked application and has substantially more features. It has also been heavily battle-tested over the years and still exists today. Its documentation is much more detailed, easy to use, and understand than mitmproxy. Using Wireshark's packet inspection tools, debuggers could effectively debug the networked data they need during a connected session without fear of the debugging session compromising remote connections. A standard debugger could be used in parallel with Wireshark to debug local application issues while still inspecting any networked data.

# Scenario 3: Bug on Target Platform Without Debug Tools

## Description of Issue/Scenario.

The core issue with this scenario is that bugs are occuring on only a specific target platform where the **debugger doesn't have access to the same debugging tools used on a development platform**, effectively preventing them from using these tools to debug the issue since it only occurs on the target platform.

# Details of Issue/Scenario

The root of this type of issue is inherent to platform specific differences between the target platform and the development platform; e.g, if a game is developed on Windows x64 with a target platform on a game console. The bugs are happening **only when the app is running on the target platform** ( in this case, a console), where the Windows debugging tools are not available to inspect what is causing the issue. Assuming target platform dev tools are not provided (e.g console dev kits), this makes debugging the target platform-only issues incredibly difficult without access to the original development platform debugging tools. The issue is not a matter of the bugs themselves, but rather the **inability for the bugs to replicate on the development platform** and the **lack of access to debugging tools on the target platform**.

What should be happening in this scenario is that the debugger should have some way to debug a version of the target platform build on their development platform using either the same debugging tools for both platforms or different tools for each platform. Whatever the case, the target platform build should be debuggable with a debugger on the development platform instead of only being runnable on the target platform hardware itself.

# Discovery/Replication of Issue

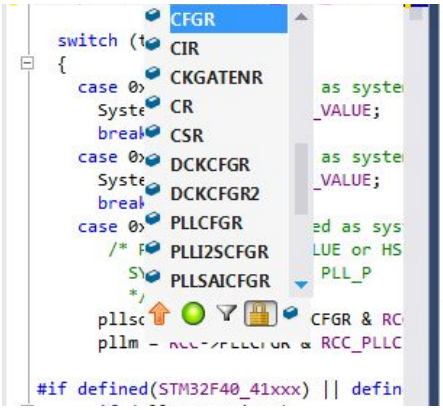
Issues with platform specific bugs themselves are not the problem here, but rather the fact that the debugger has no access to solve these bugs when the application is running on the target platform. Because of this, the bugs will always be replicated without the ability to debug the target platform build on the development platform; since the bugs only occur on the target platform, any time the target platform build is run on the target platform, the bugs will happen. This means that replication is essentially guaranteed.

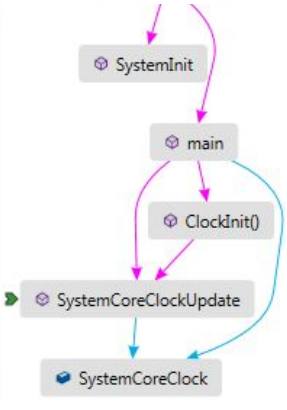
# Tool Research/Strengths and Weaknesses

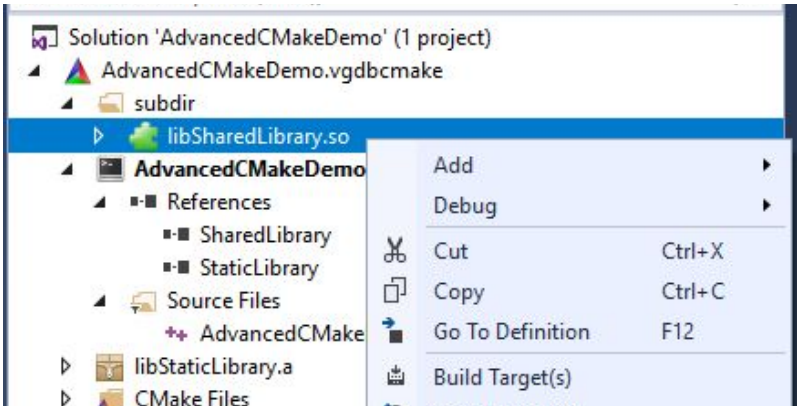
Since this issue is related not to the code of an application itself but rather access to the debugging tools, a debugger would need to make use of **a multiplatform debugging tool** while the program is running on the development platform with a build for the target platform. With these tools, debuggers can debug issues on their target platform while working on their development platform seamlessly, allowing any platform specific bugs to be analyzed, tested, and ultimately solved.

Based on my research, I've identified the following two tools that are best suited for this sort of issue.

## VisualGDB




An cross-platform development debugging tool for Visual Studio that supports a variety of different target platforms and is designed specifically for debugging multi platform projects

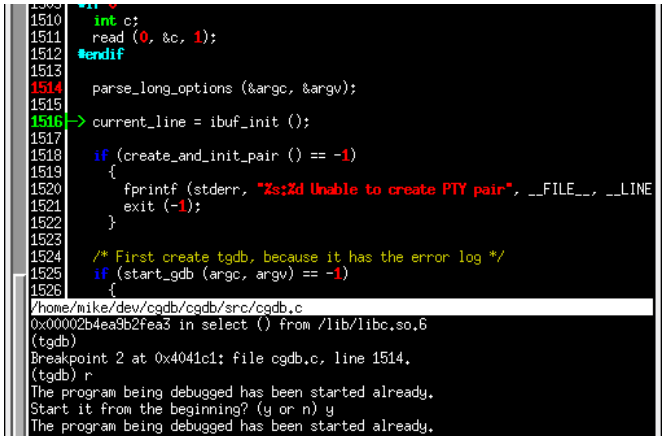
Strengths:

- Setup to work with a variety of platforms
- Built specifically for debugging and to work with Visual Studio
- Customizable debugging environments for specific platforms
- Supports a wide variety of platforms and devices for platform specific debugging
- Solid documentation

Weaknesses:

- Paid software
- Complicated interface with a lot of features
- Focused mainly on software dev, may not work on certain platforms (e.g game consoles)

## [GDB Debugger]

A rather standard and classic debugging tool similar to Visual Studio's debugger with a multitude of other features for cross platform development that make it ideal for this scenario.
Strengths:

- Allows for remote debugging on cross platform environments
- Features standard debugging tools like breakpoints and watches
- Can debug programs running on the same machine, other machines of different platforms, or through simulators and virtual machines
- Supports Windows, Mac, and Linux
- Widely used in various industries; tried and tested program
- Free and open source

Weaknesses:

- Can be difficult and obtuse to setup for multi platform debugging without a lot of experience
- Uses a command line interface only, unless a third party GUI is used
- Documentation is complex and hard to read

## Solution Proposal

Based on the findings from the tool research process, I propose the following solution.

Similarly to the networking scenario, finding tools tailored specifically to this situation was rather difficult; it's rather telling that both of them are GDB based (well, one is *literally* GDB itself). However, both of the tools would work for debugging on a target platform from the development platform, but would take an extensive amount of prior research of their documentation and a lot of setup to get running effectively. Once setup though, the results would be worth the effort.

In terms of which tool to use, VisualGDB is by far a more user friendly program since it features a rather streamlined GUI and has many of the same features of GDB itself already embedded within. Debuggers could use VisualGDB to hook into a build for the target platform on their development platform and trigger breakpoints on the target platform specific bugs using the tool's session window and catch points. That streamlined interface and robust feature set comes with a price point however, making this tool unattractive for students or small companies. For more experienced developers, using GDB itself would be a preferable route; its open source and heavily modified to suit any development platform context. After some setup and modifying, GDB can be configured to hook into essentially any target platform from the development platform thanks to it's remote debugging capabilities.