

University of Denver

Digital Commons @ DU

---

Electronic Theses and Dissertations

Graduate Studies

---

1-1-2016

## PECCit: An Omniscient Debugger for Web Development

Zachary Ryan Azar  
*University of Denver*

Follow this and additional works at: <https://digitalcommons.du.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Azar, Zachary Ryan, "PECCit: An Omniscient Debugger for Web Development" (2016). *Electronic Theses and Dissertations*. 1099.

<https://digitalcommons.du.edu/etd/1099>

This Thesis is brought to you for free and open access by the Graduate Studies at Digital Commons @ DU. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons @ DU. For more information, please contact [jennifer.cox@du.edu](mailto:jennifer.cox@du.edu), [dig-commons@du.edu](mailto:dig-commons@du.edu).

PECCit:  
An Omniscient Debugger for Web Development

---

A Thesis  
Presented to  
the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science  
University of Denver

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science

---

by  
Zachary R. Azar  
March 2016

Advisor: Professor Matthew J. Rutherford

© Copyright by Zachary R. Azar 2016

All Rights Reserved

Author: Zachary R. Azar  
Title: PECCit: An Omniscient Debugger for Web Development  
Advisor: Professor Matthew J. Rutherford  
Degree Date: March 2016

## **Abstract**

Debugging can be an extremely expensive and time-consuming task for a software developer. To find a bug, the developer typically needs to navigate backwards through infected states and symptoms of the bug to find the initial defect. Modern debugging tools are not designed for navigating back-in-time and typically require the user to jump through hoops by setting breakpoints, re-executing, and guessing where errors occur. Omniscient debuggers offer back-in-time debugging capabilities to make this task easier. These debuggers trace the program allowing the user to navigate forwards and backwards through the execution, examine variable histories, and visualize program data and control flow. Presented in this thesis is PECCit, an omniscient debugger designed for backend web development. PECCit traces web frameworks remotely and provides a browser-based IDE to navigate through the trace. The user can even watch a preview of the web page as it's being built line-by-line using a novel feature called capturing. To evaluate, PECCit was used to debug real-world problems provided by users of two Content Management Systems: WordPress and Drupal. In these case studies, PECCit's features and debugging capabilities are demonstrated and contrasted with standard debugging techniques.

# Acknowledgements

First, I would like to thank my advisor, Dr. Matthew Rutherford. Dr. Rutherford was not only a fantastic advisor, but a great mentor and friend. His patience, reassurance, and guidance were invaluable. I would also like to thank my parents, family, friends, and girlfriend. Their continual support and love during these years at the University of Denver made this accomplishment possible. Lastly, I would like to thank the other members of my oral defense committee, Dr. Chris GauthierDickey and Dr. Davor Balzar, for their time and feedback.

# Contents

<b>1</b>	<b>Introduction and Background</b>	<b>1</b>
1.1	Background: Traditional Debugging Methods . . . . .	3
1.1.1	Log-Based Debugging . . . . .	3
1.1.2	Debugger/IDE Debugging . . . . .	4
1.2	Background: Software Failures . . . . .	5
1.2.1	A Software “Bug” . . . . .	5
1.2.2	Fixing Software Failures . . . . .	8
1.2.2.1	Understanding the Code . . . . .	8
1.2.2.2	Localizing the Root Cause . . . . .	9
1.3	The Shortcomings of Modern Debuggers . . . . .	9
1.4	Should Developers Be Debugging Differently? . . . . .	12
1.5	Omniscient Debugging . . . . .	14
1.6	Introducing PECCit . . . . .	15
<b>2</b>	<b>Related Work</b>	<b>17</b>
2.1	Omniscient Debuggers . . . . .	17
2.2	Replay-Based Debuggers and Tracing . . . . .	24
2.3	Reverse-Executing Debuggers . . . . .	33
2.4	Query-Based Debuggers . . . . .	36
2.5	Fault Localization and Automated Debugging . . . . .	39
<b>3</b>	<b>PECCit</b>	<b>42</b>
3.1	PHP, Web Pages, and Frameworks . . . . .	43
3.2	Automated Debug Server (ADS) . . . . .	44
3.3	PECCit Session Manager . . . . .	46
3.3.1	Managing Sessions . . . . .	47
3.3.2	Changing Settings . . . . .	49
3.3.3	System Status . . . . .	50

3.3.4	Sending Commands . . . . .	50
3.4	PECCit Inspector . . . . .	50
3.4.1	Step Navigation . . . . .	51
3.4.2	File Navigation and Execution Path Highlighting . . . . .	54
3.4.3	Variable Pane, Variable Inspection, and Variable Differencing . . . . .	55
3.4.4	Query Info Pane . . . . .	58
3.4.5	Search Tool . . . . .	60
3.4.6	Step Finder Pane . . . . .	62
3.4.7	Capturing . . . . .	66
3.5	Implementation . . . . .	69
3.5.1	Automated Debug Server . . . . .	70
3.5.1.1	Xdebug . . . . .	70
3.5.1.2	ADS Design and Workflow . . . . .	72
3.5.1.3	PECCit Settings . . . . .	74
3.5.1.4	Database and Session Storage . . . . .	77
3.5.1.5	Capturing . . . . .	78
3.5.1.6	Language Independence . . . . .	80
3.5.2	PECCit Web Interface . . . . .	81
3.5.2.1	Handling the Data . . . . .	82
3.5.2.2	PECCit Session Manager Implementation . . . . .	83
3.5.2.3	PECCit Inspector Implementation . . . . .	84
<b>4</b>	<b>Evaluation and Analysis</b> . . . . .	<b>87</b>
4.1	Case Study 1: Non-Admin Can Upgrade Database . . . . .	89
4.1.1	Background . . . . .	89
4.1.2	Problem . . . . .	90
4.1.3	Setup . . . . .	91
4.1.4	Using PECCit . . . . .	93
4.1.5	Analysis . . . . .	94
4.2	Case Study 2: Missing Logo on Theme . . . . .	98
4.2.1	Background . . . . .	98
4.2.2	Problem . . . . .	98
4.2.3	Setup . . . . .	100
4.2.4	Using PECCit . . . . .	100
4.2.5	Analysis . . . . .	103
4.3	Case Study 3: Duplicate Stores In Plugin . . . . .	105
4.3.1	Background . . . . .	105
4.3.2	Problem . . . . .	106

4.3.3	Setup . . . . .	106
4.3.4	Using PECCit . . . . .	108
4.3.5	Analysis . . . . .	115
4.4	Case Study 4: Incorrect View Count Plugin . . . . .	116
4.4.1	Background . . . . .	116
4.4.2	Problem . . . . .	116
4.4.3	Setup . . . . .	118
4.4.4	Using PECCit . . . . .	118
4.4.5	Analysis . . . . .	128
4.5	Case Study 5: Capitalized Titles in Drupal Theme . . . . .	130
4.5.1	Background . . . . .	130
4.5.2	Problem . . . . .	130
4.5.3	Setup . . . . .	130
4.5.4	Using PECCit . . . . .	131
4.5.5	Analysis . . . . .	139
4.6	Case Study Analysis . . . . .	139
<b>5</b>	<b>Conclusion</b>	<b>142</b>
5.1	Future Work . . . . .	143
5.1.1	Performance Improvements . . . . .	144
5.1.2	Additional Features . . . . .	144
5.1.3	Language Independence Improvements . . . . .	145
	<b>Bibliography</b>	<b>146</b>



# List of Figures

1.1	Software Infection (Source [Zel05]) . . . . .	7
3.1	PECCit System Overview . . . . .	45
3.2	PECCit Session Manager: Sessions Table . . . . .	47
3.3	PECCit Session Manager: Additional Features . . . . .	48
3.4	PECCit Session Manager: Launching the Inspector . . . . .	48
3.5	PECCit Session Manager: Changing Settings . . . . .	49
3.6	PECCit Session Manager: Sending Commands . . . . .	51
3.7	PECCit Inspector . . . . .	52
3.8	PECCit Inspector: Step Tree . . . . .	53
3.9	PECCit Inspector: Labeled with Variable Pane Open . . . . .	54
3.10	PECCit Inspector: Navigation Buttons . . . . .	55
3.11	PECCit Inspector: Source Code File Select . . . . .	56
3.12	PECCit Inspector: Variable Inspect Tool . . . . .	58
3.13	PECCit Inspector: Inspection Results . . . . .	59
3.14	PECCit Inspector: After \$v Has Been Set to 14 . . . . .	60
3.15	PECCit Inspector: Example of Variable Differencing . . . . .	61
3.16	PECCit Inspector: Query Info . . . . .	62
3.17	PECCit Inspector: Search Tool for Variable Name . . . . .	63
3.18	PECCit Inspector: Search Tool for Variable Value . . . . .	64
3.19	PECCit Inspector: Search Tool with Inspect Results . . . . .	65
3.20	PECCit Inspector: Jumping to Step From Search/Inspect Results . . . . .	65
3.21	PECCit Inspector: Step Finder . . . . .	67
3.22	PECCit Inspector: Step Finder from Step Tree . . . . .	68
3.23	PECCit Inspector: Capture Showing Incomplete Web Page . . . . .	68
3.24	PECCit Inspector: Using Chrome DevTools with Capture . . . . .	69
3.25	PECCit Workflow Diagram . . . . .	73
4.1	Case Study 1: Forum Post . . . . .	92

4.2	Case Study 1: Update Screen . . . . .	93
4.3	Case Study 1: Session Manager . . . . .	95
4.4	Case Study 1: First Session . . . . .	96
4.5	Case Study 1: Second Session . . . . .	97
4.6	Case Study 2: Forum Post . . . . .	99
4.7	Case Study 2: Preview With Image . . . . .	100
4.8	Case Study 2: Home Page Without Image . . . . .	101
4.9	Case Study 2: Header File . . . . .	102
4.10	Case Study 2: Step Finder . . . . .	103
4.11	Case Study 2: Modification Settings Before Fix . . . . .	104
4.12	Case Study 2: Home Page With Image . . . . .	104
4.13	Case Study 2: Modification Settings After Fix . . . . .	105
4.14	Case Study 3: Forum Post . . . . .	107
4.15	Case Study 3: Test Plugin with Test Store . . . . .	108
4.16	Case Study 3: Sessions . . . . .	109
4.17	Case Study 3: Store Search . . . . .	111
4.18	Case Study 3: Step Finder . . . . .	112
4.19	Case Study 3: \$store_data Array . . . . .	113
4.20	Case Study 3: Variable Inspector Tool . . . . .	113
4.21	Case Study 3: Results of Variable Inspector . . . . .	114
4.22	Case Study 3: \$stores Array . . . . .	114
4.23	Case Study 4: Forum Post . . . . .	117
4.24	Case Study 4: Home Page with Incorrect Formatting . . . . .	118
4.25	Case Study 4: Test Page with Correct Formatting . . . . .	119
4.26	Case Study 4: Sessions . . . . .	120
4.27	Case Study 4: Search Tool Results for "12" . . . . .	121
4.28	Case Study 4: Inspect Tool in Search Results . . . . .	122
4.29	Case Study 4: Results of Inspecting \$results->today . . . . .	122
4.30	Case Study 4: Results from pvc_get_stats() of Home Page . . . . .	123
4.31	Case Study 4: Differences Tool on Two Lines . . . . .	123
4.32	Case Study 4: Results from pvc_stats_counter() of Home Page . . . . .	124
4.33	Case Study 4: Results from pvc_stats_show() of Home Page . . . . .	125
4.34	Case Study 4: Using Step Finder . . . . .	125
4.35	Case Study 4: Results from pvc_stats_show() of Page . . . . .	126
4.36	Case Study 4: How the Home Page versus the Test Page Retrieves Content in the Theme . . . . .	126
4.37	Case Study 4: Using the Normal Version Instead of Excerpt Version on the Home Page . . . . .	127

4.38	Case Study 4: Initialization of Hooks . . . . .	128
4.39	Case Study 4: Home Page with Correct Page Count Formatting . . . . .	129
4.40	Case Study 5: Forum Post . . . . .	131
4.41	Case Study 5: Capitalized Content . . . . .	132
4.42	Case Study 5: The PECCit Session . . . . .	134
4.43	Case Study 5: Choosing the File in Step Finder . . . . .	134
4.44	Case Study 5: Before Page Print . . . . .	135
4.45	Case Study 5: After Page Print . . . . .	136
4.46	Case Study 5: \$page Variable . . . . .	136
4.47	Case Study 5: First Capture of Unstyled Front Page . . . . .	137
4.48	Case Study 5: Second Capture of Unstyled Front Page . . . . .	137
4.49	Case Study 5: Chrome DevTools Showing CSS Properties . . . . .	138
4.50	Case Study 5: Chrome DevTools With Deselected CSS Property . . . . .	138

# Chapter 1

## Introduction and Background

Debugging is an extremely important facet of software engineering. The process can be quite frustrating though, often requiring more time and energy than developers would like to devote. One study found that debugging can take almost 50% of a developer's time [LVD06]. This lost time results in lost money. According to a study by the National Institute of Standards and Technology, the national cost of inadequate testing and debugging was estimated to be \$59.5 billion [Tas02]. The study argued that improvements to the infrastructure could reduce this cost by nearly \$22.2 billion.

Debugging can be difficult with standard debugging techniques because of the way bugs infect a system. To find a bug, the developer typically needs to navigate backwards through infected states and symptoms of the bug to find the initial defect. To assist with this process, most developers have settled with either log-based debugging or breakpoint debuggers with an Integrated Development Environment (IDE) [SPTH14]. Though these

techniques can work, they are unable to help the developer work backward through infected states. One strategy that is underutilized is *Omniscient Debugging* [Lew03].

Omniscient debugging, also known as *back-in-time* or *reverse* debugging, allows a developer to debug forward and backward in time within the same execution trace. It achieves this by tracing the program as it's running. Once done tracing, the user can step forward/backward, search through variables and values, query about variable histories, and ultimately learn a lot more about the execution than a standard debugger.

Presented with this thesis is PECCit, an omniscient debugger designed for web developers. PECCit traces web frameworks and provides a browser-based IDE which the user can use to move bidirectionally through a trace. PECCit can provide variable histories, arbitrary access through system states, variable/value searching, and execution path highlighting. PECCit also provides a novel feature called *capturing* which allows the user to watch the web page as it's being built line-by-line. PECCit is designed to be language independent such that further improvements could extend the tool for other web development languages.

PECCit was used to debug real-world problems through case studies. The problems were taken from support forums for the two most common Content Management Systems: WordPress<sup>1</sup> and Drupal<sup>2</sup>. In these case studies, PECCit's features and debugging capabilities are demonstrated and contrasted with standard debugging techniques.

---

<sup>1</sup><https://wordpress.org/>

<sup>2</sup><https://www.drupal.org/>

## 1.1 Background: Traditional Debugging Methods

Debugging is typically performed using a combination of two tactics. The first is log-based debugging. This is when the developer inserts some code that either prints information to the console or output buffer or logs the information to a file or other form of storage. The second traditional strategy for debugging is using a debugger or Integrated Development Environment (IDE) which offers breakpoint debugging. This allows the user to step through the execution of the code, stop at conspicuous places where a bug may be, and analyze variables and the call stack. The following sections examine these two in more detail.

### 1.1.1 Log-Based Debugging

Log-based debugging is excellent for quickly gaining information and solving small bugs but has major problems with scalability and practicality when bugs get even slightly more complex. With log-based debugging, the developer can quickly add a print statement to the code to learn more about a variable or about the code execution. For example if a C++ developer wanted to learn more about the variable  $x$  at a particular moment in time, they could insert `std::cout<<"x is " <<x <<std::endl;` into their code to print the variable to the screen. The user must then re-execute the code and watch for the printed variable. This is a quick strategy and works fine for small problems but doesn't scale.

Log-based debugging is unable to scale because logs get too large, the user needs to guess when and what to print, and re-execution could be slow or impossible. Printing a single variable can be done quickly but printing all variables in scope (which may be

necessary to debug the problem) quickly makes the logs massive and unreadable. This also alters the code as these log statements need to be inserted throughout the code. Log statements might be repeated over and over if the print is in a loop or long-running program. The log can get too large to walk through and it can sometimes be difficult to match log messages with the line of code / time of execution that printed the log. The user also needs to guess where to insert the log messages and what to print. If they guess wrong, the needed information to find/fix the bug might not be printed and the user will need to re-execute and try again. Each time, these re-executions could require quite a bit of time (especially if there is a lot of logging) and maybe the re-execution is impossible like if the execution is time-sensitive or if the bug is difficult to reproduce due to non-deterministic execution (like unpredictable input/output).

### **1.1.2 Debugger/IDE Debugging**

A more sophisticated solution than log-based debugging is using a debugger or Integrated Development Environment (IDE). These tools allow developers to stop the execution of a program using breakpoints and often allow the user to inspect the variable values and call stack at that paused moment in time. Once an execution is halted, many of the tools allow the user to step forward in time to see how the remaining lines of code affect the variables and execution. With tools like *Step Over*, *Step Into*, and *Step Out*, the user is able to navigate forward in time while watching the variables and execution. For web

development, common debuggers/IDEs include Visual Studio<sup>3</sup>, NetBeans<sup>4</sup>, Eclipse<sup>5</sup>, and PHPStorm<sup>6</sup>.

Debuggers/IDEs have been the standard for debugging practices for years as they can be very effective tools for finding bugs. Suppose a developer wanted to see what value is assigned to a variable  $x$  in this Java code `int x = findAllEntries();`. They could set a breakpoint at this line and see what value is assigned to  $x$  during the execution. Knowing this, the developer could make a hypothesis as to where a bug is occurring. These debuggers are common tools in industry and education but they have major limitations due to their forward-in-time nature. Before examining these shortcomings (see Section 1.3), the following section examines how a bug enters a program and how developers fix these bugs.

## 1.2 Background: Software Failures

This section discusses software bugs (also known as infections), how they damage an execution, and how they can be fixed by understanding the code and localizing the root cause.

### 1.2.1 A Software “Bug”

A software bug is an infection in the program[Voa92]. Zeller explains that a program failure caused by a bug can be visualized in four steps[Zel05]:

---

<sup>3</sup><https://www.visualstudio.com/>

<sup>4</sup><https://netbeans.org/>

<sup>5</sup><https://eclipse.org/>

<sup>6</sup><https://www.jetbrains.com/phpstorm/>



**The programmer creates a defect**

This is the root cause where the programmer writes a piece of code (consciously or inadvertently) that ultimately causes the failure.

**The defect causes an infection**

The defective code is executed in such a way that the “the program state differs from what the programmer intended.”[Zel05]

**The infection propagates**

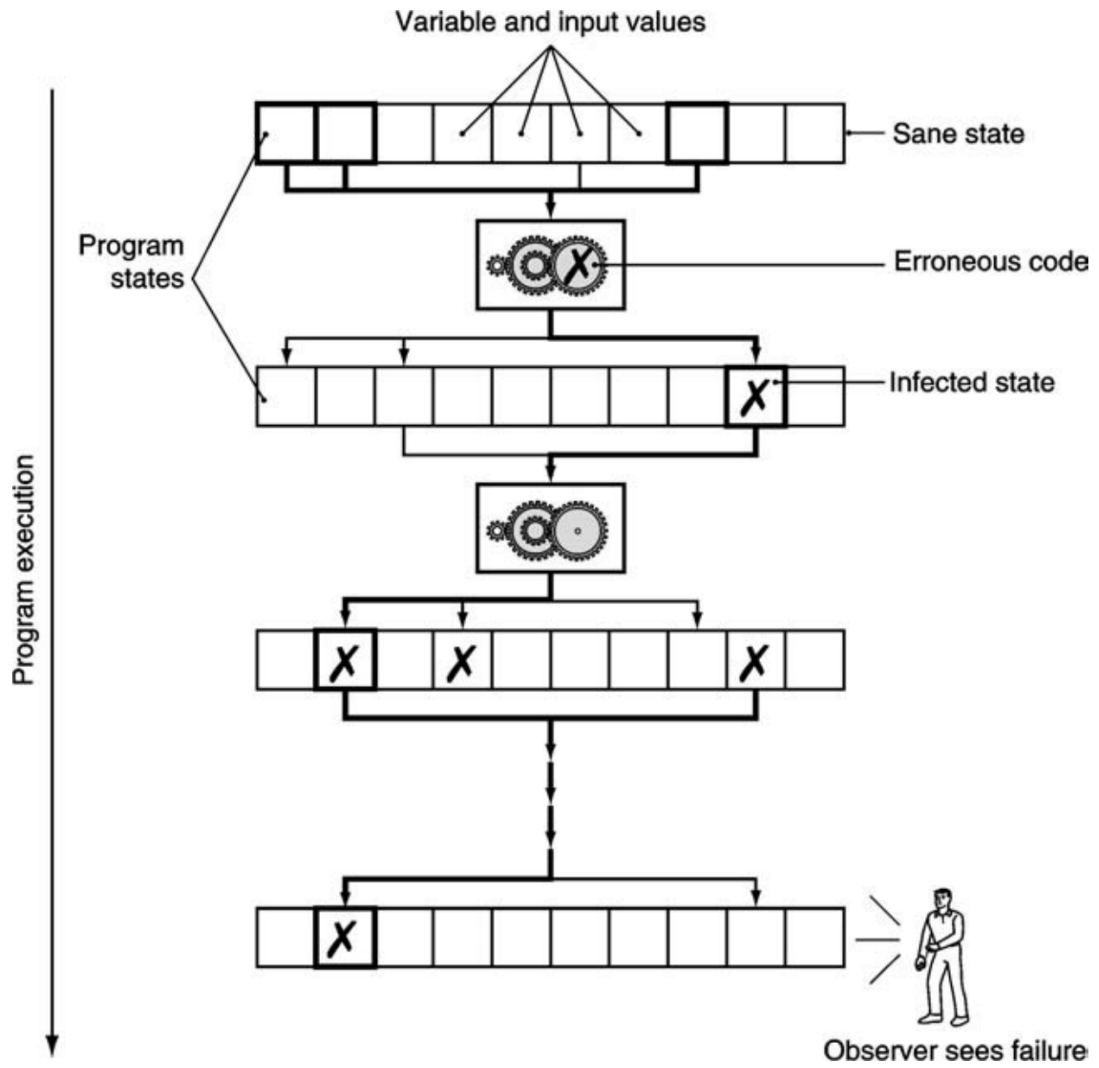
The infection could cause other infections. It could get masked or hidden. In unexpected ways, the infection moves throughout the execution and program states.

**The infection causes a failure**

The infection causes a noticeable error to the developer or user. This is the symptom of the initial defect and could be directly caused by the defective code or indirectly from another place that the infection had spread.

A software infection can be visually interpreted as Figure 1.1 from [Zel05]. The program most likely starts from a correct/sane state. Then, the execution hits code that contains a defect. This is the “bug.” This code might erroneously cause an infection that could change a program state to something that the developer wasn’t intending. This state and erroneous code could propagate through the execution (even when used with correct code) to infect other program states. Some of these states could be masked or not even noticed. Finally, the developer or user observes a failure.

**Figure 1.1:** *Software Infection* (Source [Zel05])



## 1.2.2 Fixing Software Failures

Whether developers are creating a new program or maintaining a shipped product, programmers ultimately want to fix software failures quickly and in their entirety. To fix a software failure, the developer first needs to make a mental model of the program's behavior and how the source code is creating that behavior. Next, the developer must localize where the defect is in the code. Finally, the developer must fix the bug and test. Though important, this thesis will not discuss in detail fixing and testing software bugs as this is outside of the scope of the research presented. The following subsections look at these first two steps and how tools can help developers.

### 1.2.2.1 Understanding the Code

First, the developer must make a mental picture of what they believe the code is doing and what it's actually doing. Often, developers will start by searching through the code (manually or using a search tool) to find relevant code that could be causing or demonstrating the unintended behavior [KMCA06]. Through this navigation through the code (sometimes while the program is actually executing using a breakpoint debugger), the developer is learning more about the behavior and the system state and how that compares to the intention of the program and the intentional system state. This process has been called *Bridging the Gulf of Evaluation* [ND86]. Once the developer understands the purpose of the code and where system states are differing from their intended behavior, they can start to try to find the bug.

### 1.2.2.2 Localizing the Root Cause

Next, the developer moves through these infected system states trying to localize the root cause. This step is called *localization* when the developer must look throughout the source code/states to find the initial defect [LF95]. What makes this non-trivial is that not only does the developer need to search through the code (space) but they must also search through the system states as they change through time [CZ05]. This process can be demonstrated with Figure 1.1 where the developer finds one of the lower (later in time) infected states and works backward to find that initial defect. This backward movement and dependency connections can also be understood as a cause-effect chain [Zel02] where the developer realizes that A caused B which caused C etc. Once the developer has moved back far enough, they can isolate and fix the defect.<sup>7</sup> Development tools like debuggers help to find bugs but they cannot move backward through space and time.

## 1.3 The Shortcomings of Modern Debuggers

The major reason traditional debuggers/IDEs are not the ideal tool for debugging is their forward-in-time nature. As discussed in Section 1.2.2.2, the developer typically starts at the symptom of the bug or observable failure and works their way back up the infection to find the root cause. The developer must understand the forward direction of the cause and effect chain, but typically they ask questions in the backward direction like “why was this variable

---

<sup>7</sup>Once the defect is fixed, the question remains whether this was truly the root cause. For example from Figure 1.1, the developer might have changed code which fixes one of the intermediate infected states but the developer might not have found the initial defect. There is another field of research involving software testing to determine if other bugs exist in the program. Software testing and bug prevention is not examined in this thesis

set to this value?” or “why did this code execute?”. Traditional debuggers are only able to move forward. Thus, the developer must make a guess as to where the bug occurs and pick a time before the defect so that they can move forward into it. They set a breakpoint at this location and re-execute. If they’re lucky/intuitive and they did pick a point before the defect, then they could still miss the bug if they *Step Over* where the error occurs. In this scenario, the developer stepped too far and must again re-execute (and potentially pick a new breakpoint). This is not a scalable solution as sometimes these programs need to run for quite some time before the bug occurs or breakpoint breaks. Also, a breakpoint might be part of a long loop in which case the developer might need to manually step forward quite a few times to find the bug. This process is error-prone as it only takes one step too many to overstep and require another re-execution.

Some modern debuggers have advanced tools which can help the developer set better breakpoints like conditional breakpoints and data breakpoints [Zel05]. Conditional breakpoints allow the developer to say “break when  $x$  equals 5.” Data breakpoints (also called watchpoints) allow the user to say “break whenever  $x$  changes.” These advanced breakpoints can be quite powerful if the developer knows which variables/values are important to help find the bug. The fallbacks of these tools are that they can result in much slower execution, the developer needs to know which variables/values to watch before starting the execution, and the developer is still able to under-step (go too far back resulting in too many steps required to find the bug) or overstep (step past the bug requiring the developer to try again and re-execute).

Another weakness of modern debuggers is their temporary and limited awareness of information during the execution. At a breakpoint, the debugger tells the user of the current

scope of variables/values and the current call stack. The debugger has no knowledge of what happened prior to the breakpoint and no intuition of what will come after. As soon as the user takes another step, the information of the previous state is gone. If the user wants to compare the two steps, the developer must write down the information or save it elsewhere which is error-prone and time consuming. Also once the debugging session is over, nothing is saved or remembered from the execution. A developer can't ask the debugger "was function *foo()* ever called?" or "what values were assigned to *x* throughout the execution?" once the debugging session is over. Most developers would agree that being able to ask these questions would be an extremely handy feature of modern debuggers.

Since everything is forgotten after the debugging session concludes, the developer is unable to discuss or share their experience with other developers. To do so, the developer would need to write down their results or take screenshots (both of which are not very helpful for other developers). One study found that the most frequent source of information for a developer is her coworkers [KDV07].

Along the same lines, modern debuggers have a difficult time with bug reproduction. Often, it's difficult to reproduce a bug even if a developer is given clear instructions on how to do so. The same study found that some developers had such a hard time reproducing a bug that they would often just create a remote-desktop connection to the bug report author's computer so they could see the bug [KDV07]. Modern debuggers do not help with this task. Also if the bug is hard to recreate (perhaps due to non-deterministic qualities like unpredictable input/output), then that forgotten debugging session might have held the key to finding the defect but now it's gone and can't be reproduced. In the modern age where

software teams work remotely and software has gotten more and more complex, a scalable and shareable solution must exist.

## **1.4 Should Developers Be Debugging Differently?**

Debugging remains to be one the most tedious and time consuming activities that a developer faces even with modern debuggers [Zel05]. A study published in 2008 reported that 72% of companies that they surveyed admitted that their debugging process was “problematic” [Bal08]. Interestingly though, the study noted that 62% of the companies said that their “defect management and testing approach either ‘did not require improvement’ or that it wasn’t possible to create change in their approaches (despite problems)” [Bal08]. The same study reported that 37% of developer time was spent toward debugging. Of the companies reported, 67% reported that it takes 2-10 workdays to fix bugs and 11% said that it takes 11-30 days. It is obvious that debugging could be improved, but perhaps companies do not know how to improve it or that they believe modern debuggers are as good as they’re going to get.

Debugging strategies could clearly be improved, but deciding on the qualities of the next-generation debugger is not particularly straightforward. When looking over various debugging stories, Eisenstadt examined what made the bugs in the stories difficult, what their root cause was, and how the developers eventually solved them [Eis97]. He found that “more than 50% of the difficulties are attributable to just two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools inapplicable.” For the inapplicable tools category, the developers would find that

having debugging activated would mask the bug or that they couldn't recreate it (or the debugging configuration removed the bug). Also in the study, he suggests the ideal tool that could help the developers who participated.

“We have identified a niche that really needs attention; the most heavily populated cell in our three-dimensional analysis suggests that a winning tool would employ some data-gathering or traversal method for resolving large cause/effect chasms in the case of memory-clobbering errors” [Eis97]

Thus, Eisenstadt's recommended tool could potentially gather data about the execution, let the developer traverse quickly through the execution, illuminate cause-effect chains, and potentially help with variable initialization and value changing.

Through Eisenstadt's research [Eis97] along with the research about how bugs infect a system [Voa92, Zel05] and how bugs are fixed using a mental model [KMCA06, ND86] and localization [LF95, CZ05, Zel02], we can begin to identify qualities of a next-generation debugger that could lead to more efficient debugging. The next-generation debugger would ideally move forward *and* backward through time and space allowing the user the ability to move backward through the infection. The ideal tool could quickly answer questions like “was this code executed?” and “what are all of the values that were assigned to this variable?”. The tool could help us make a better mental model of the program and behavior. Also, it should be more scalable and sharable in that it should reduce the number of re-executions and allow the user to save and share debugging sessions. This would be especially helpful if the bug is difficult to recreate or the program is long running. Interestingly enough, all-seeing debuggers have been available for quite some time which offer all of these features. They are called *Omniscient Debuggers*.



## 1.5 Omniscient Debugging

Omniscient debuggers offer a multitude of powerful capabilities to the developer. The term *Omniscient Debugging* was first coined by Bil Lewis in 2003 [Lew03] but the concept has been around for years [Bal69]. Omniscient debuggers typically function by performing a trace of the execution as it runs. Everything that happens during the execution (i.e. a function is called, a variable is initialized, a variable's value is changed, etc.) is saved and the developer is able to traverse forward and backward through this trace after the execution has completed. The developer can move through the different lines of code that were executed, examine the various variable states, and often ask questions like “was this function called?” and “what values were assigned to this variable?”.

Despite being built for many different languages and having alternatives like query-based debugging and replay-based Debugging (see Chapter 2), omniscient debugging and similar tools have not been widely adopted by industry nor education [SPTH14].<sup>8</sup> One reason could be the performance drawbacks. While it's being traced, program execution can be quite slow. Also, these traces can get large and require lots of space. Though researchers have made considerable efforts in making these traces smaller and executions more scalable [PTP07, LGN08, BM14], performance overhead can quickly intimidate a developer. However the time required to perform the trace, leading to a quick debugging session to find the bug, might be less than the total time required to debug the program using a traditional debugger.

---

<sup>8</sup>Bil Lewis, a major proponent for omniscient debugging, even expresses his surprise/concern about the lack of interest in omniscient debugging on his website <http://www.lambdacs.com/debugger/>.

Ultimately, omniscient debugging could be the missing strategy that greatly reduces debugging time for software developers. It naturally allows for backward traversal of execution steps to find infected system states. Users can quickly jump through time (execution steps) and space (source code) to better understand the code and localize the root cause of the infection. Though expensive in resources, these tools often provide settings to the user which can greatly reduce overhead and trace size. A software field where omniscient debuggers are lacking however is in backend web development (see Chapter 2).

## **1.6 Introducing PECCit**

Presented in this thesis is PECCit, an omniscient debugger designed for backend web development. PECCit traces web frameworks remotely and provides a browser-based IDE to navigate through the trace. The user can step forward and backward through the trace and access arbitrary locations instantly. PECCit provides variable histories allowing the user to see when variables were created and changed throughout the entire execution. The user can also search through these variables and values and determine what changes were made to system state across lines of code. PECCit also has execution path highlighting allowing the user to quickly see which files and lines of code were used in the execution. Additionally, PECCit offers a novel feature called capturing which allows the user to watch a preview of the web page as it's being built line-by-line. PECCit is controlled and used entirely in the browser allowing for team collaboration and scalability. Resource requirements and overhead can be decreased using various PECCit trace settings.

The remainder of this thesis is structured as follows: Chapter 2 discusses related work including other omniscient debuggers as well as other debugging alternatives to standard debuggers/IDEs; Chapter 3 presents PECCit's various features and outlines how PECCit was implemented; In Chapter 4, PECCit is evaluated through the use of case studies and its successes and weaknesses are discussed; Lastly in Chapter 5, the thesis is concluded with a discussion of current debugging practices as well as improvements that could be made to PECCit.

# Chapter 2

## Related Work

There are a number of other omniscient debuggers and debugging strategies in both the commercial and academic sectors [Eng12]. They include omniscient debuggers, replay-based debuggers, reverse-executing debuggers, query-based debuggers, and fault localizing and automated debuggers. Each of these strategies have the same goal: to make debugging easier and more efficient. They differ in their implementation and provided features. The following sections discuss these different strategies and how they compare to PECCit.

### 2.1 Omniscient Debuggers

Multiple omniscient debuggers exist in industry/academia but there are subtle differences in their implementation and functions that make them all unique. The majority of these debuggers use an event driven strategy which is different from PECCit's implementation. Typically in the event driven strategy, code is inserted into the program either when it is loaded into a virtual machine or into the byte code at runtime. This code will report back

to the omniscient debugger when important events occur like a function call or variable change. These events are timestamped and displayed to the user after the execution in such a way that the user can see everything important that happened during the execution of the program. This is different from PECCit in that no code is inserted using PECCit.<sup>1</sup> Instead of waiting for an important event, PECCit is constantly logging what is going on at every line of code. This makes PECCit slower in that it's not native code, but much more versatile because it is not language/virtual machine dependent like the debuggers in this section. Other tools have to insert code that is specific to that language. PECCit simply needs to talk with a debugger that is capable of interrupting an execution and reporting variable values (see Section 3.5 for more information on the implementation of PECCit and Section 3.5.1.6 on language independence).

The concept of omniscient debugging dates back to 1969 with the publication of EXDAMS (EXtendable Debugging And Monitoring System) [Bal69]. EXDAMS was an impressive contribution with a number of features. It enables the user to scroll forward and backward in time (after execution) through statements and variable values, analyze errors, and *flowback* through data to see where values are derived by creating an inverted value tree. It works by analyzing the code statically and building a model of the program and control functions. During this phase, it also inserts debug statements into the code specific to EXDAMS which is then used during runtime to build a "history tape" which it then uses to playback the execution to the user. These features are powerful, but the system assumes that the compiler doesn't dramatically change the code and that the source code is in a compatible language so that it can insert appropriate debug statements.

---

<sup>1</sup>Except if capturing is enabled. When enabled, the program inserts lines of code during the execution to retrieve the current output buffer. See Section 3.5 for more details.

An early omniscient debugger which works with a subset of C is PROVIDE, a “Process Visualization and Debugging Environment” [Moh88]. While their interpreter executes the code, PROVIDE records all states in a database. Users are able to start the debug session before the program has finished and indirectly make queries on that database to understand more about the execution and state changes. While they are navigating, users can move forward, backward, and to arbitrary states. Along with other features, PROVIDE was one of the first successful visual tools for debugging. With PROVIDE, functions, variables, and other components are shown visually to the user and the user manipulates his view to understand more about the state changes and interactions. The tool only works with a subset of C however and only supports integers, characters, and one-dimensional arrays.

Another older implementation of an omniscient debugger is ZStep95 for the Lisp programming language [LF95, ULF97]. The interactive tool is designed to help programmers watch how static code is run dynamically, both in the forward and reverse directions. The user is able to use the Graphical User Interface (GUI) to run and edit Lisp code and visually examine bugs, code flow, and errors. The researchers focused heavily on the user interface/experience of the GUI and put most of their attention into how they could best display the data from the underlying omniscient debugger to help the user understand the execution enabling them to find bugs.

One of the first and most prominent omniscient debuggers for Java was presented by Bil Lewis with his implementation called ODB [LD03, Lew03]. With this publication, he also coined the name *Omniscient Debugging*. The implementation is open-source and available online<sup>2</sup>. The tool is Java specific and works by inserting code into the Java classes

---

<sup>2</sup><http://www.lambdacs.com/debugger/>

as they are loaded into the JVM. The inserted code fires events when something important happens, like variable alteration or a function call. ODB provides a GUI with multiple panes but can also be used as an Eclipse<sup>3</sup> or NetBeans<sup>4</sup> plugin. Thus, the tool could be used for web development if the language used is Java, but it is not specifically designed for web development. The tool is very customizable and is able to handle complex functionality like debugging multithreaded programs and complex filtering searches like “when does  $foo(x,y)$  get called with parameters  $x=13$  , $y=20$ ?”. Like other implementations, ODB is event driven where an event triggers ODB code which marks a timestamp and alters internal data structures to keep track of variable changes and code control. In 2006, Bil was invited to talk at Google TechTalks about ODB and omniscient debugging.<sup>5</sup>

Another important Java implementation is TOD, a “Trace-Oriented Debugger” presented by the University of Chile [PT09, PTP07]. TOD functions similarly to ODB as it is event based and inserts code into the classes as they load in the JVM. TOD aims to outperform ODB in terms of scalability. Instead of storing the events and program structure in memory as ODB does, TOD stores events and program structure into separate databases. Their system allows for more scalable performance with parallelized and distributed qualities. TOD is capable of complex queries and is able to tone back performance statically (pick which classes to record) and dynamically (manually turn on recording during runtime). TOD can work as a standalone application or an Eclipse plugin.

An option for Squeak, a SmallTalk dialect, is Unstuck released in 2006 [HDD06]. Similar to other event based systems, Unstuck injects code into the Squeak byte code during

---

<sup>3</sup><https://eclipse.org/>

<sup>4</sup><https://netbeans.org/>

<sup>5</sup><https://www.youtube.com/watch?v=xpl8hIgOyko>

runtime. The code fires events when methods are called, when they return, and when variables are altered. These events are stored during runtime and a trace is built out of these events once processed. The trace information is then used to recreate state and a standalone GUI which allows the user to move around the execution and do simple searching, variable highlighting, and object back-tracing.

Lienhard, Gîrba, and Nierstrasz also offer a solution in Squeak which could be extended to other languages with object-oriented virtual machines [LGN08]. They recognize the power behind omniscient debugging but also note that the slowdown during execution as well as the memory overhead can make the practice impractical. Thus, they provide a model for saving the execution information in the form of object aliases and support the performance benefits with a Squeak implementation. Anytime an object is referenced (created, copied, destroyed, etc) an alias is created acting like a middle man between the reference and the actual object. This alias is in charge of remembering information about the history of the object. For example, suppose there is a *Person* object *p* with a *name* field. That *name* property will point to an alias which points to the actual object (a string). Then when the name is changed, the alias will remember the old name and update with the new name. When execution is stopped, the user can look back through the history in the aliases to see everything that happened. Also, these aliases are smartly garbage collected as the various objects die (unless they were used as the target of a method call or as a parameter). Therefore, not all variables and events remain in memory. This makes the implementation faster and more space efficient. However, this could also mean that a needed variable for debugging could have gone out of scope and deleted. The authors do not make an attempt to understand/keep all dependent variables as they chose to forgo this feature for speed.



In addition to these speedups, all aliasing is done in memory so debugging is efficient but heavily memory consuming.

A currently available commercial, omniscient debugger is UndoDB by Undo Software<sup>6</sup>. UndoDB supports C and C++ on Linux machines and Android devices. Once installed, it replaces all of the functionality of *gdb*<sup>7</sup> but claims to add much, much faster process recording (see Section 2.3 for information on *gdb*'s process recording). As it has similar commands to *gdb*, it can be used on the command line or integrated into common developer tools like Emacs<sup>8</sup> and Eclipse. UndoDB is similar to PECCit in that it is a full omniscient debugger which allows arbitrary access and variable inspection. Like PECCit, the user can save the sessions and send/share them with other developers. PECCit is different in that it targets web developers (offering web specific tools like capturing), addresses a different programming language (PHP), and provides its own GUI.

Another “Historical Debugger” is the IntelliTrace feature built into Microsoft Visual Studio<sup>9</sup>. IntelliTrace is a new feature in the latest 2015 Enterprise Edition and it works with VB, C#, ASP.NET, Microsoft Azure, Windows Forms, WCF, WPF, etc. It does not support C++.<sup>10</sup> This is a commercial feature as the Enterprise Edition is quite expensive<sup>11</sup>. It doesn't trace everything but it records exceptions, .NET Framework calls, and function calls (arguments and return values). The function call tracing is not enabled by default as it can incur quite a bit of overhead.<sup>12</sup> With IntelliTrace, the user can filter modules that

---

<sup>6</sup><http://undo-software.com/>

<sup>7</sup>*gdb* is the standard C/C++ debugger on a linux machine. More information can be found at <https://www.gnu.org/software/gdb/>

<sup>8</sup><https://www.gnu.org/software/emacs/>

<sup>9</sup><https://www.visualstudio.com/>

<sup>10</sup><https://msdn.microsoft.com/library/dd264915.aspx>

<sup>11</sup><https://www.visualstudio.com/products/visual-studio-enterprise-vs>

<sup>12</sup>[https://msdn.microsoft.com/library/dd264915.aspx#Anchor\\_4](https://msdn.microsoft.com/library/dd264915.aspx#Anchor_4)

they're interested in, save sessions to a file, and search through historical data. It can also monitor and record apps running on other servers or in production. There is even an API hookup to the IntelliTrace system allowing for programmatic control. Though IntelliTrace works for C# and ASP.NET (and thus can debug web applications), it is not web focussed like PECCit so it doesn't offer features like capturing and automatically getting the query info. It's also built into a larger and more expensive enterprise framework.

Diver is an omniscient debugger that focusses on displaying run-time behavior to the user [MS10]. The tool only saves execution sequence information like function calls (it does not save variable and state information). Research with the tool focuses on user understanding of the code using the various views and sequence diagrams. It is provided as a set of plugins for Eclipse.

Expositor is an interesting academic contribution as it works on top of UndoDB's commercial omniscient debugger [KFH13]. It allows the user to write scripts to make queries on the UndoDB debugger. These scripts can programmatically navigate around the trace and quickly find information. Expositor offers internal data structures to navigate the various time/state snapshots of UndoDB to offer features like mapping, filtering, and scanning. Though these features are powerful, there could be a learning curve as developers will need to learn how to write these scripts including the API and classes to use Expositor. This could be an excellent solution for advanced developers who need to debug complex problems on C/C++ (and who have purchased UndoDB).

Another powerful, commercial omniscient debugger available for Java is Chronon.<sup>13</sup> The Chronon system uses two tools: the Recording Server and the Time Traveling Debug-

---

<sup>13</sup><http://chrononsystems.com/>

ger. The Recording Server records Java applications. It boasts minimal overhead and is designed for long-running programs. Recording can be stopped/started dynamically and controlled remotely using a web application. Recording sessions can be saved, shared, and automatically created/flushed. The creators report excellent performance which they attribute to static analysis of the code. The analysis yields predictions which can be used to limit the amount of recording to non-obvious sections of code (like non-deterministic actions)<sup>14</sup>. The Chronon Time Traveling Debugger (Eclipse plugin) is used to examine these recordings and offers execution path highlighting, variable inspection, arbitrary jumping, as well as many other features. PECCit shares many of these features, though PECCit is aimed at web development with features like capturing and a browser-based debugger.

## 2.2 Replay-Based Debuggers and Tracing

To combat the overhead of omniscient debugging, a new strategy was born based on replaying the execution. Some researchers argued that maintaining full traces of programs might not always be necessary to debug programs and offer back-in-time debugging. With a traditional debugger if the user wants to break at a point farther back than their current paused location, the user sets a breakpoint at that location farther back in the code and re-executes the program. Replay-based debuggers work in this same way. When the user wants to travel back-in-time, the debugger will re-execute the program automatically and break earlier for the user. There are three main hurdles for replay-based debuggers: long-running programs, non-determinism through input/output (I/O) and race conditions, and offering variable histories.

---

<sup>14</sup><http://chrononsystems.com/blog/chronon-3-recorder-internals>

Long-running programs can make replay-based approaches unpractical. If the user wants to step only a few statements back in time, the debugger would need to re-execute the entire long-running program to offer that. Thus, most replay debuggers offer *checkpointing*. With checkpointing, the debugger will make periodic checkpoints throughout the first execution. At these checkpoints, the debugger will take a snapshot of the system state. Then when the user wants to go back in time, the debugger will return to one of these checkpoints and re-execute from the checkpoint instead of re-executing from the beginning of the program. This can save lots of time at the cost of periodic resource use during the first execution. One problem that can still disrupt the re-execution is non-determinism though.

Non-determinism in a program can cause major issues for replay-based debuggers since they can only function properly if the re-execution perfectly matches the initial execution. If they didn't match, the system state or execution path from the back-in-time re-execution might not be the same as the first execution so the user could have difficulty trying to debug an infected state from the first execution. Non-determinism can be caused by I/O (file contents could change, or the user could click a different button), race conditions (perhaps the program is multithreaded and the thread CPU scheduling was slightly different in the re-execution), or even system calls. Some of the replay-based approaches check for non-determinism and if it could exist, warn the user and become inoperable. To handle I/O, others record everything coming in and out of the program so that re-executions can be faithfully recreated. To handle multithreading, others use tools (like [CS98]) to recreate exact thread scheduling.

The last hurdle that some debuggers have is variable histories (if they want to offer that feature). This feature allows the user to ask “what are all of the values that were assigned to  $x$ ?” or “break at the last place that  $x$  changed.” With this first question, the answer is similar to that of watchpoints from Chapter 1 where the replay-based debugger replays the program and watches  $x$ . The second question is more difficult because the debugger doesn’t know the *last* place  $x$  was changed prior to the current break point without rerunning and reaching the current breakpoint. In this scenario, some replay-based debuggers actually replay the execution twice. The first time,  $x$  is watched like a watchpoint and the debugger writes down every moment in time  $x$  changes. Once the execution reaches the current break point, the program is re-executed (again) until it reaches the point it wrote down the final change to  $x$  and pauses the debugger here for the user. With PECCit, finding the last change of  $x$  is nearly instant since everything is recorded the first time. However, the first execution of replay-based debuggers is much faster than omniscient debuggers like PECCit since replay-based debuggers only perform small maintenance work and not full tracing.

Thus, replay-based debuggers offer similar features to omniscient debuggers like PECCit but they use different strategies (with different pitfalls). During execution, the replay-based debuggers are much faster. However, that speed improvement might be lost during the debugging stages when the user wants to travel back-in-time or access variables multiple times. PECCit and other omniscient debuggers take an upfront “let’s record everything in case we need it” approach to debugging. Replay-based debuggers take a “let’s wait and see what the user wants, then perform various re-executions if the user needs more information” approach. Replay-based debuggers have to worry about things like long-running programs and I/O which are not as big of an issue for omniscient debuggers. Long-running

programs for omniscient debuggers typically require lots of resources, though this can be controlled by the proper use of settings or periodically releasing data. I/O and threading information is saved the first time for omniscient debuggers so there is no need to perfectly recreate them since omniscient debuggers rely on traces instead of actual execution for re-execution and replay. With proper trace settings, variable history information is nearly instant with omniscient debuggers since this data is stored. Replay-based debuggers do not have this luxury (though this makes their first execution much faster since they don't need to store anything). The remaining portion of this section discusses some of the replay-based debuggers throughout academia and industry.

One of the first replay-based debuggers was COPE [ACS84] published in 1984. The COPE system uses checkpointing to reduce re-execution time and it recycles space by deleting old blocks (traces of the execution). This limits the system in how far back it can recover, but reduces space requirements.

Another older replay-based debugger was IGOR which was published in 1988 [FB88]. The prototype worked for C programs. It also uses checkpointing to cutdown re-execution time. The system is capable of changing out code dynamically and even attempts to handle the difficulties presented from I/O operations.

An early publication of debugging parallel programs using a replay-based approach is Recap [PL88]. Also published in 1988, Recap was designed to periodically make checkpoints of a multithreaded program while it's running. The user is able to select one of these checkpoints and start execution from that point. The tool works by acting as a middle-man for all system calls and memory handles. Also, it periodically asks all code to stop and perform a checkpoint. At this checkpoint, the code forks a new process and suspends it.

Thus if the user wants to replay execution from a checkpoint, the suspended process at that checkpoint is unsuspending and the logged signals/system calls are used from the original execution in the replay.

Another solution for parallel programs in C was RecPlay [RDB99]. This tool was designed with practicality in mind by placing time consuming computation in the replay stage and making recording of the program extremely fast. This allowed the user to always have recording on and only use replaying when needed. During the replay phase, the system performs on-the-fly data race detection and informs the user of any non-deterministic activity or race conditions. It takes the stance that these data races are bugs and informs the user of these conditions. If no race is detected, the system guarantees a correct re-execution.

bdb was another early replay-based debugger [Boo00]. It was published in 2000 and works for C and C++. bdb uses checkpointing to cutdown on re-execution time. It also records all I/O from system calls so that re-executions are faithfully recreated.

Another published replay-based debugger is Reverse Watchpoint [MT03]. Though prototypes were made in C, the major contribution was written for Java. Using a byte code transformer, Reverse Watchpoint inserts code into classes which the user would like to debug. The tool allows the user to ask questions like “when was variable  $x$  last changed?”. It does this by using the strategy described earlier in Section 2.2. The first re-execution watches the variable  $x$  for all changes, then the next re-execution breaks on the last place that it was changed. The authors note that their tool has low overhead since they are using byte injection. However, Reverse Watchpoint does not have solutions for the common replay-based pitfalls like long-running programs and non-determinism.

Another solution that works on Linux systems is Jockey [Sai05]. With Jockey, there is no need to change the source code or run the code in a special engine. Jockey is a shared object file that runs as part of the target application. Jockey takes hold of the process and intercepts I/O and other non-deterministic actions that the program takes. Thus, it can record what happens and accurately replay an execution. Since Jockey runs on the same process as the target application though, problems arise if the application is malicious with memory. Jockey and the target application are part of the same process and share all resources. Thus, Jockey must take precautionary measures so that it does not affect the target process nor be affected by it. Since everything is logged, Jockey is capable of replaying non-deterministic programs allowing the user to debug the program using replay-based approaches.

Created by a number of researchers at Microsoft, Nirvana is a run-time engine which can provide back-in-time debugging [BCdJ<sup>+</sup>06]. As an engine, Nirvana sits on top of the operating system and programs can run on the engine. When configured, Nirvana takes control of applications and threads running on it and inserts code into the execution. The code inserted is used to create a highly compressed trace file which can be fed back to the engine to simulate the execution of the program. Using checkpointing and the trace file, the researchers claim that they built back-in-time functionalities into a debugger, though their primary research goals appear to be the Nirvana framework and another framework called iDNA. Back-in-time debugging is more of a complimentary feature of the tracing.

A more recently published back-in-time debugger using replay-based strategies is Tardis [BM14]. This publication presents some of the generic algorithms for performing replay-based debuggers, then explains the implementation and features of Tardis. Though the



algorithms could be used generically in any managed runtime environment (like Java, JavaScript, etc.), Tardis uses the .NET Common Language Runtime (CLR)<sup>15</sup> to hook into the compiler and replace code which will add hooks for the debugger, record variable states, and intercept environment interactions like memory allocation, I/O, and thread scheduling. Tardis uses highly optimized/compressed checkpoints (snapshots) to keep overhead low and uses interesting strategies like full tracing during the forward execution after a re-execution from a checkpoint to improve debugging experience.

Following Tardis, Mark Marron, PhD<sup>16</sup> appears to be porting some of the strategies to Microsoft's next-generation Microsoft Edge Browser<sup>17</sup>. The browser's developer tools include a time-traveling debugger which offers "interrogative virtualization." Similar to TARDIS, the system is replay based with checkpoints, recorded events and non-deterministic actions, and tracing on checkpoint replay. The browser, along with this back-in-time feature, have not been released yet. Though the solution targets web development like PECCit, it appears to mainly focus on frontend development (JavaScript in the browser once the page is already built) whereas PECCit addresses the need for a backend debugger while the page is being built. It also doesn't appear to offer a capturing feature.

QueryPoint is another tool that can be used for frontend web development as it is designed for JavaScript [MBP11b, Mir12]. It is an implementation of the *lastChange* algorithm [MBP11a] which is a replay-based strategy. QueryPoint is a Firefox plugin that interacts with Firebug<sup>18</sup>, the primary debugging tool for Firefox. With execution paused at a breakpoint, the user can ask QueryPoint to find the *Last Change* of a variable or object

---

<sup>15</sup>[https://msdn.microsoft.com/en-us/library/8bs2ecf4\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx)

<sup>16</sup><http://research.microsoft.com/en-us/um/people/marron/>

<sup>17</sup><https://channel9.msdn.com/blogs/Marron/Time-Travel-Debugging-for-JavaScriptHTML>

<sup>18</sup><http://getfirebug.com/>

property. QueryPoint will re-execute the program, saving information about the system state every time the variable is changed. Once the re-execution hits the paused point, the information from the last time the variable was changed is returned to the user. The authors report their tool is highly efficient and has an edge over other replay-based debuggers because it doesn't have to worry about non-determinism in re-executions (as long as the bug is reproducible) because of the way that the tool re-executes and saves state information.

Another recently published frontend web tool is Timelapse with Dolos [BBKE13]. Dolos is a system capable of recording execution in JavaScript using Webkit<sup>19</sup>. Once the session is recorded, Timelapse is able to visually walk through the replays with timelines, events, bookmarks, etc. Dolos is reported to be fast and scalable while offering deterministic re-execution using I/O logging. Admitting that there is still a possibility of non-determinism, the authors note that Dolos can actually warn the user if it suspects infidelity in the re-execution. What's also interesting is the authors performed a small study with developers to see if their tool improved debugging. The authors didn't find a significant time difference when debugging programs when compared to traditional tools, but they found that expert developers were better able to incorporate the tool into their debugging strategies while the tool seemed to distract the less experienced developers.

Related to replay-based debuggers, other researchers have tried to identify how to make these tools better. Netzer and Weaver examined how to efficiently checkpoint long-running programs [NW94]. They present an adaptive approach to tracing a program such that it can be re-executed from incremental checkpoints. They present optimal and approximation algorithms which allow a system to adapt to the running program and decide what to trace

---

<sup>19</sup>A web engine used in Chrome and Safari. See <https://webkit.org/>

and how often to trace. Xu, Rountev, Tang, and Qin examined how to make checkpoints more efficient [XRTQ07]. They present a strategy for statically analyzing the code. Their analyzer determines what to trace and what to ignore which can be inferred later during replay using control-dependence-based slicing.

Other replay-based tools exist that address entire systems. King, Dunlap, and Chen published their time-traveling virtual machines which can debug operating systems [KDC05]. Simics was released in 1998 as a commercial tool which was a full system virtual platform [MCE<sup>+</sup>02, EAW10]. Simics could simulate a full system (including multiple computers, processors, files, network, devices, running different programs, etc.) and offered checkpointing and replaying. These tools, though quite powerful, address a very different debugging space than the user level (specifically, backend web development space) that PECCit addresses.

A related field of research just focuses on performing full traces of systems and programs [ZG05]. The strategies which these tools use to perform the traces dictate how they can best be used for debugging (and a number of other applications). The WET tracing strategy intertwines the data and control flow histories of the execution allowing for easier reverse execution and debugging [ZG05]. The Tralfamadore system traces low-level machine execution which is excellent for tracing operating system code and kernels [LCH<sup>+</sup>12]. Once a trace has been performed, the user can use Tralfamadore to make queries on the traced execution to better debug and understand the code's behavior. Thus, this tool could also be categorized as a query-based debugger (see Section 2.4). Queries could include “show me a histogram of all parameters ever passed to function *foo()*” and “what are the common paths of packets through the network stack”. An older publication

presented the TRAPEDS system which focussed on fully tracing multicomputers [SF89]. Though the primary focus is tracing, each of these tools could be used for debugging in a non-traditional way using replay-based techniques.

## 2.3 Reverse-Executing Debuggers

Another solution for back-in-time debugging is reverse-executing debuggers. These debuggers offer back-in-time navigation, but in a linear fashion. They execute in the reverse direction by *undoing* each execution statement. This is quite different from omniscient debuggers. Since omniscient debuggers trace entire executions, they typically offer instant access to arbitrary locations in the execution. Reverse-executing debuggers simply execute in both directions. Thus if the user wants to access a point much farther back than the current paused position, the reverse-executing debugger will reverse, step-by-step, back to that point. Reverse-executing debuggers still offer powerful, back-in-time functionality but at the cost of execution overhead as well as the time required to execute backwards to reach the desired location.

One of the earliest reverse-executing debuggers was presented in 1971 [Zel73, Zel71]. This tool, called Retrace, was built for PL/I and was a function added to the language. The programmer could reverse the order of execution, but they had to write code to do it programmatically. Thus, the execution couldn't be interacted with while it was running.

Another reverse-executing debugger is Cornell's Program Synthesizer [TR81]. Published in 1981, this full-fledged IDE offered many features still offered today by modern IDEs including code templates, a full screen GUI with a tree and text editor, and a diag-

nostic interpreter to find problems. The program also offered a back-in-time debugger. The debugger used basic reverse-execution strategies by storing system changes and restoring them with each reverse step.

Another reverse-executing debugger was Spyder [ADS91]. This tool was quite powerful as the user could stop execution (or have it stop on an error), travel in the reverse direction, change some of the code, and then resume execution after the change. To move backward, the tool remembers a series of *change-sets* which log all of the necessary information to essentially undo each line of code. The tool also offers Dynamic Program slicing to help determine which statements/variables could have an affect on a given variable (more on Dynamic Slicing in Section 2.5).

Another tool that attempts to handle reverse execution for a symbolic debugger was presented by Cheng, Fuchs, and Chung [CFC01]. Their tool functions by inserting code into a copy of the program code during compilation which logs old variable values into a history buffer during execution. Since there are two versions of the code, the user can enable/disable debugging in real time. Also, the user can specify which subroutines to record and the compiler will pick which subroutine to run (original or instrumented). In reverse order, the values are removed from the history buffer. To save space, the buffer is a wraparound so reversibility is limited by space.

A major visual contribution is the Java Interactive Visualization Environment (JIVE) [RR05, GJ04, GJ05].<sup>20</sup> JIVE is a standalone tool or Eclipse Plugin [CJ07] that allows the user to visualize the execution and changing system states of a Java program. It uses the Java Platform Debugging Architecture (JPDA) to receive important events from the

---

<sup>20</sup><http://www.cse.buffalo.edu/jive/>

execution and creates two models: the object model and the sequence models. After the execution with these models and their interactions, JIVE presents the execution (sequence and objects) in diagrams similar to UML so that the user can learn more about and debug the execution. The user is able to step back (using reverse-stepping by undoing state changes) to watch as the sequence diagrams and object diagrams change. The user can also search through variable histories. The tool is especially good at helping the user understand the Java program being debugged and could be useful for education.

Probably the most known reverse-executing debugger is provided by gdb<sup>21</sup> called Process Record<sup>22</sup>. When enabled, gdb will record the execution by logging the effects of each instruction during runtime. Then if the user wants to travel in the reverse direction, the debugger uses the logs to undo each line of code as per the user's request. The user can customize the logging behavior (linear/circular) and instruction limit. The developers at Undo Software argue that the recording is extremely slow and resource consuming though (see Section 2.1 for more information about UndoDB).<sup>23</sup>

TotalView is a commercial, reverse-executing debugger<sup>24</sup>. The tool is for C/C++ programs on Linux machines and offers a standalone IDE. The user is able to turn debugging on/off in real-time and the tool supports multithreading, I/O, distributed and network applications, etc.<sup>25</sup> It is designed for High Performance Computing (HPC) so it works with many of the common parallel frameworks/libraries including OpenMP and MPI. As it is a

---

<sup>21</sup><https://www.gnu.org/software/gdb/>

<sup>22</sup><https://sourceware.org/gdb/wiki/ProcessRecord>

<sup>23</sup><http://undo-software.com/undodb/>

<sup>24</sup><http://www.roguewave.com/products-services/totalview>

<sup>25</sup><http://www.roguewave.com/products-services/totalview/features/reverse-debugging>

commercial product, the exact details about how it records are not publicly available but the debugger appears to use a reverse-executing strategy.

## 2.4 Query-Based Debuggers

Another alternative strategy for debugging programs is query-based debugging. These debuggers allow the user to ask questions like “why did  $x$  equal  $14$ ?”, “why did this code execute?”, and “how many elements in list  $L$  have positive values?”. Some of the tools can pause execution based on these queries like “break when  $foo(x,y)$  is called with parameters  $12$  and  $4$ ”. Some of these tools require the user to write code before execution. Others can query on-the-fly. Some even come up with the questions for you. Though not classified as back-in-time debuggers, they similarly answer questions during or after execution using tracing and other methods. These queries can be quite complex involving multiple functions and lots of data. Thus, they are much more powerful than the simple conditional breakpoints provided by modern debuggers.

OPIUM is a query-based debugger for Prolog [Duc99b]. It is a trace analyzer in that as the program is running, the various events are stored as a trace in a database. The user is able to write questions/queries about these events and the debugger will pause the Prolog execution when the query is true. Interestingly since these queries are just yes or no questions, the user actually writes the queries in Prolog to debug the Prolog program.

As an adaption of OPIUM in C, COCA was created [Duc98, Duc99a]. COCA is built on top of *gdb* and the user is able to query the program state/execution using control flow information and runtime data. COCA, like OPIUM, is used on the command line, queries

are written in Prolog (with a few primitives added), and is able to conditional break execution using these advanced queries.

DUEL is a similar tool built on top of *gdb* [GH93]. DUEL uses its own, C-like, language to make queries. It allows the user to ask questions like “how many elements of list *L* have a positive value?” and “does list *L* contain two elements with identical value fields?”. Though it’s built on top of *gdb* for C programs, it’s designed to work with other debuggers and languages. One hurdle is that learning the DUEL language to ask questions could have a learning curve.

Another query-based debugging tool was released for Self, a dialect of SmallTalk [LHS97]. This tool allows users to search large object spaces and object relationships. The queries can be made on-the-fly or saved for later use. The query language is also based in Self (the authors argue that the query language should be similar to the traced language to reduce the learning curve for developers). In a query, the user specifies the search domain (where to search) and the constraints for that domain.

A query-based debugger for Java is Caffeine [GDJ02]. This tool is designed using the Java Platform Debugger Architecture (JPDA) to keep track of important events during the execution. Before the code is executed, the developer writes questions in Prolog. On-the-fly, the system runs these queries as the program is executing allowing the user the ability to ask questions about the running program. These queries can look for simple events like “how many times does *foo()* get called?” or search for complex relationships like “is this class a singleton (and truly only created once)?”. These features are powerful, but the developer must write the queries before the execution and are limited by their own mastery of Prolog.



A very unique and powerful tool is Whyline [KM04, KM08]. Originally created for Alice (an educational programming language), this now Java specific tool is capable of creating questions and answers about a program's execution . It works similarly to other event based approaches by inserting code into Java classes as they load in the JVM. However once the execution is complete (and the events and variables are stored in a trace), the tool generates questions for the developer like “why did  $x$  equal 14?” , “why did `getVal()` return 20?”, “why did this execute?”, and conversely “why *didn't* this execute?”. The tool is capable of answering these questions visually allowing the user to navigate around the different events. It uses static/dynamic program slicing and other techniques to generate and answer these questions (See Section 2.5 on program slicing). In a very small user study, the authors found that novice developers using Whyline were able to debug a particular program twice as fast as expert developers who did not have Whyline.

A similar tool to Whyline was published for one-way constraints [VZBJ04]. With the tool, the user is able to ask questions like “why did this happen?” and “Something's Wrong. Please suggest a reason”. The program uses *constraint slicing* which is a form of program slicing (see Section 2.5 for more information on Program Slicing) except that the entire dataflow is not saved reducing overhead. The slices are displayed to the user visually so that the user can understand what code and variables affect the variable in question. When the user asks questions, the tool looks through these slices and the relationships of these dependencies to suggest a problem/solution.

## 2.5 Fault Localization and Automated Debugging

As discussed in Section 1.2.2.2 of the introduction, the ultimate goal of debugging is to find the root cause of the infection (and fix it). The tools and research discussed thus far help a developer manually find these bugs but quite a bit of research has been performed on fault localizing tools which aim to find the defect for the user [WD09]. Some of the tools even suggest corrections for the defect. Though the goal of finding the bug is the same goal as that of the PECCit system, their strategy is quite different. PECCit is an interactive tool that assists the user in finding the bug. Fault localization tools use a mixture of source code analysis, execution tracing, and automation to find the bug. This section briefly discusses these tools.

A major area of research for fault localization is Program Slicing [XQZ<sup>+</sup>05]. There are different strategies for program slicing but the overall idea is that given an execution point and a subset of variables (known as the slicing criterion), program slicing is the process of finding all statements/variables that could have directly or indirectly had an impact on that subset of variables. The slices can be used for debugging, code comprehension, testing and coverage, parallelization, etc. Originally published by Weiser in 1979, static slicing examines the source code and based on all possible executions, argues which statements could affect the variable set [Wei79]. Dynamic slicing (published by Korel and Laski in 1988 [KL88]) executes the program and finds the slice based on what actually happened during that single execution.

Multiple studied slicing algorithms have been researched [XQZ<sup>+</sup>05, ZGZ03] and these strategies can be combined with other technologies to provide powerful debuggers (like the query-based Whyline debugger [KM08] or the reverse-executing Spyder debugger [ADS91]

as discussed previously). The computation and traces from slicing can yield a heavy overhead so there is lots of research on making the slicing process faster and smaller [ZG04, DPS96, WR04].

Though there are a number of other strategies and algorithms for performing fault localization through automated debugging ([Zel02, LNZ<sup>+</sup>05]), one that stands out in particular (because of its technology relation to PECCit) is presented by Artzi, Dolby, Tip, and Pistoia [ADTP10]. This research involves automated testing and debugging of dynamic web pages written in PHP. In their earlier work ([AKD<sup>+</sup>08, AKD<sup>+</sup>10]), the researchers used concrete and symbolic execution strategies to white-box test PHP applications. The testing tool, named Apollo, could even simulate user interaction. This tool could be used to find paths and interactions within the PHP framework that resulted in malformed HTML errors. The tool could show that the errors existed, but couldn't find them. Then in 2010, they combined their tool with an adaptation of the Tarantula algorithm ([JH05, JHS02]) to perform fault localization [ADTP10]. Thus, Apollo would automate thousands of tests and when HTML errors were found, the buggy execution traces were used in the Tarantula algorithm to pinpoint which lines of code most likely caused the errors. With some algorithm modifications, they showed that their tool was very successful. A few years later, a number of the same researchers published a tool which could actually repair a lot of those errors (if the errors were caused by incorrectly printing string literals – which surprisingly quite a few are) [SSA<sup>+</sup>12].

As mentioned, PECCit has a fundamentally different approach to debugging than the fault localizing/automated works mentioned in this section. Strategies such as program slicing, statistical debugging, and automated testing were created to help users find (and

sometimes fix) bugs with a more hands-off approach. The user often needs to initially interact (like setting the slicing criterion when using program slicing) but once started, the tool does the rest. With PECCit, the debugger enables the user to take control of the debugging process and provides tools and data to assist. Some argue that these automated approaches aren't very effective ([PO11]) though others have found strategies to easily find and fix small/specific bugs ([SSA<sup>+</sup>12]). The PECCit system offers benefits such as filtered traces, remote access, web page capturing, live and single recording (can debug live web traffic, and often only need to trace it once as opposed to many, many automated tests), and variable inspection which most automated debuggers and fault localizers do not offer.

# Chapter 3

## PECCit

This chapter presents the features and implementation details of PECCit: a powerful implementation of an omniscient debugger for web developers. “PECCit” was originally an acronym for *Post-Execution Code Comprehension* but its capabilities quickly outgrew just allowing the user to understand the code. Its design was intended for web developers using PHP (as there was a need for an omniscient web debugger as seen in Chapter 2). However, PECCit could be used with other languages as it doesn’t rely on language specifics or code injection (see Section 3.5.1.6 on language independence). PECCit allows the user to trace an execution, move forward and backward in time while examining the execution path, inspect variable histories, and even watch as a web page is built line-by-line.

PECCit is a system of programs that all function together to offer an omniscient debugger. It uses a number of languages and technologies including C++, MySQL, PHP, HTML, CSS, JavaScript, jQuery, AJAX, a REST API, Bootstrap, etc. More information on PECCit’s implementation can be found in Section 3.5. From a high-level view, PECCit is

an execution tracer for PHP web frameworks and it provides a browser-based debugger to explore the traces. The three major components of the PECCit system are the Automated Debug Server (ADS), the PECCit Session Manager, and the PECCit Inspector. The combined system offers a powerful and customizable omniscient debugging experience for a web developer at any experience level. Before discussing the three major PECCit components, the basics of PHP web pages and frameworks will be covered.

### 3.1 PHP, Web Pages, and Frameworks

PHP<sup>1</sup> is a scripting language that is commonly used for web development. It is open-source and allows developers to quickly create dynamic web pages. It can be embedded directly into HTML and the latest version (PHP 5) allows Object Oriented Programming. As of July 16th, 2015, PHP is used by approximately 81.9% of the top 10 million sites worldwide.<sup>2</sup> PHP code, often mixed with HTML, is interpreted, executed, and the output is then sent back to the user. The most common interpreter is the Zend Engine<sup>3</sup>.

When a user requests a PHP web page (like <http://zachazar.com/index.php>), the PHP interpreter will build and return a web page back to the user. This PHP page might have HTML in it and it might include/link other PHP files as well. Often, site administrators will use Content Management Systems (CMS) and other frameworks written in PHP. These frameworks are typically open-source, community supported, and provide common features that most sites need like user and content management, content types, and easy appearance customization through themes. The most popular CMS frameworks are Word-

---

<sup>1</sup><https://www.php.net/>

<sup>2</sup>[http://w3techs.com/technologies/overview/programming\\_language/all](http://w3techs.com/technologies/overview/programming_language/all)

<sup>3</sup><https://www.zend.com/en/community/php>

Press<sup>4</sup> and Drupal<sup>5</sup> and they are both written in PHP. These frameworks can get quite large and there is often a steep learning curve for a new developer to understand how these frameworks build the web pages and manage the data. Thus, debugging these frameworks can often be difficult, especially for new developers. The following sections examine the three main components of the PECCit system and show how they can help a developer debug web pages and frameworks.

## 3.2 Automated Debug Server (ADS)

The Automated Debug Server (ADS) is the backend of the PECCit system and is responsible for tracing the executions. When the PHP interpreter is executing the code, it interacts with an extension called Xdebug<sup>6</sup>. Xdebug can stop the execution and report information including variables/values, call stack, etc. The ADS is a software system written in C++ which communicates with Xdebug. Through this interaction, it learns everything about the execution and saves it to a MySQL database. More technical information about Xdebug, the communication between the ADS and Xdebug, and how the information is stored can be found in Section 3.5.1.

The ADS, database, Xdebug interaction is demonstrated in Figure 3.1. When a user (the laptop in the figure) requests a web page like *index.php* which might use multiple PHP files, the ADS will interact with the PHP engine through Xdebug as it builds the page. During these interactions, the ADS writes down everything that happens including which lines of code were executed and all variables and values. The ADS can even *capture* the page as

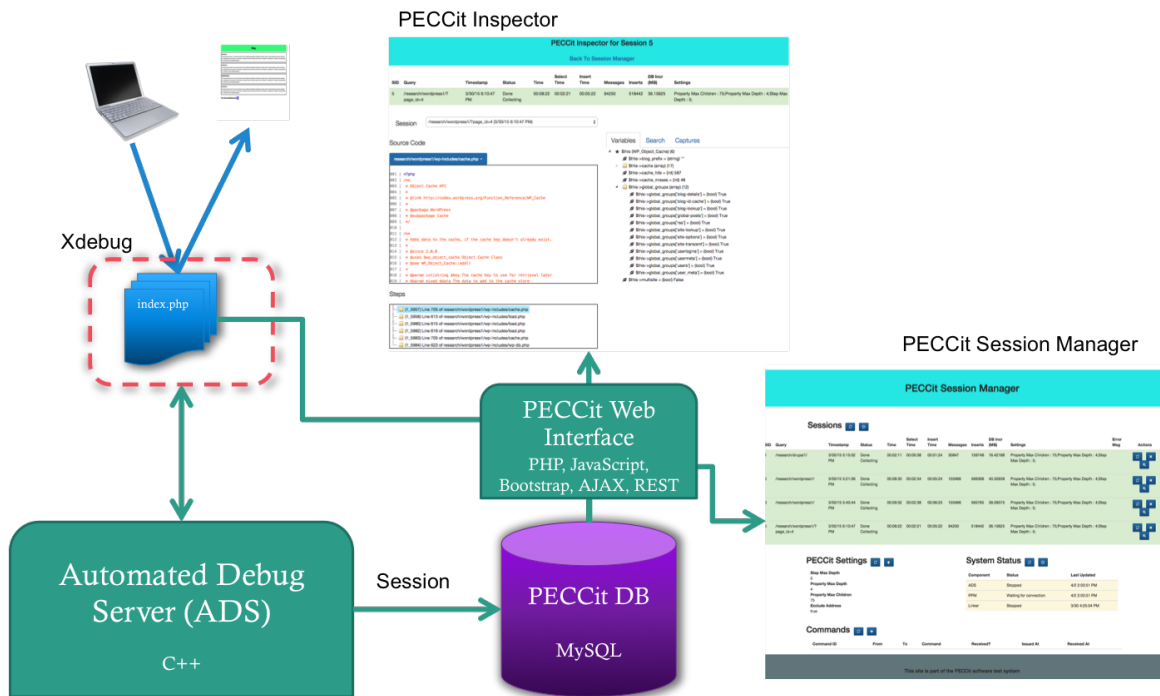
---

<sup>4</sup><https://wordpress.org/>

<sup>5</sup><https://www.drupal.org/>

<sup>6</sup><http://xdebug.org/>

**Figure 3.1: PECCit System Overview**





it's being built to show what the page looks like at each line of code. Once the execution is complete, the debugging information is saved as a *session* and stored in a database. All of this is done on the server of the web framework that is being debugged.<sup>7</sup>

The user can set various settings which affect how the ADS traces the executions. These settings include how deep to step into a trace, how many children of a variable are stored (for an array or object), which files to trace (called a whitelist), and to enable/disable web page capturing (when the ADS saves previews of what the page looks like as it's being built). A detailed list of the settings offered are provided in Section 3.5.1.3. Once sessions are saved into the database, they are managed using the PECCit Session Manager.



### 3.3 PECCit Session Manager










The PECCit Session Manager (see Figures 3.2 and 3.3) is a web application that allows the user to manage the saved sessions, manage settings for the PECCit system, view the system status, and send commands to the ADS. These various features are outlined in the following sections. The page is built using HTML, CSS, and JavaScript and more information about how the application is implemented can be found in Section 3.5.2.2. As shown in Figure 3.1, the PECCit Session Manager interacts with the PECCit database through the web interface.

---

<sup>7</sup>As PECCit is a standalone component and talks to the debug engine over the network, it doesn't necessarily have to run on the server containing the test framework. See Section 3.5 for more details.

**Figure 3.2: PECCit Session Manager: Sessions Table**



Sessions  

SID	Query	Timestamp	Status	Steps	Time	Select Time	Insert Time	Messages	Inserts	DB Incr (MB)	Settings	Error Msg	Action
1	/research/Blog/	12/20/15 3:01:36 PM	Done Collecting	60	00:00:02	00:00:02	00:00:00	554	2798	0.25000	Capture_On : true;Exclude Address : true;Property Max Children : 75;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : /;		  
2	/research/casestudies/wp/wpThe...	12/20/15 3:01:50 PM	Done Collecting With Errors	546	00:00:09	00:00:02	00:00:07	3200	18478	1.86040	Capture_On : true;Exclude Address : true;Property Max Children : 75;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : /;	ADS received a command to shutdown or cancel.	  
3	/research/casestudies/wp/wpThe...	12/20/15 3:02:45 PM	Collecting	55382	00:00:25...								  



### 3.3.1 Managing Sessions

The primary purpose of the PECCit Session Manager is to manage and handle sessions. The user can see which sessions have completed, check their analytics data, and delete them (see Figure 3.2). The row color indicates the status of the session (with green meaning complete and successful, red meaning complete but with errors, and blue meaning in progress). At the top of the sessions table, the user can refresh the session list manually (with the refresh button) or set the table to refresh itself regularly (stopwatch button). On the side of each row, the user can refresh the stats for the session (reload icon) and delete the session (X icon). By clicking the magnifying glass icon (see Figure 3.4), the user can launch the PECCit Inspector for that session (see Section 3.4).



**Figure 3.3:** PECCit Session Manager: Additional Features

**PECCit Settings**  

Step Max Depth  
50  
Property Max Depth  
4  
Property Max Children  
75  
Exclude Address  
true  
Capture\_On  
true  
Whitelist  
nowhitelist  
Site  
<http://www.zachazar.com>

**System Status**  

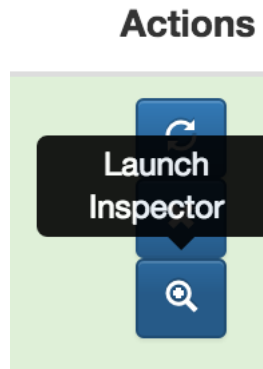
Component	Status	Last Updated
ADS	Running	12/20 3:02:45 PM
PPM	Waiting for ADS	12/20 3:02:45 PM

**Commands**  

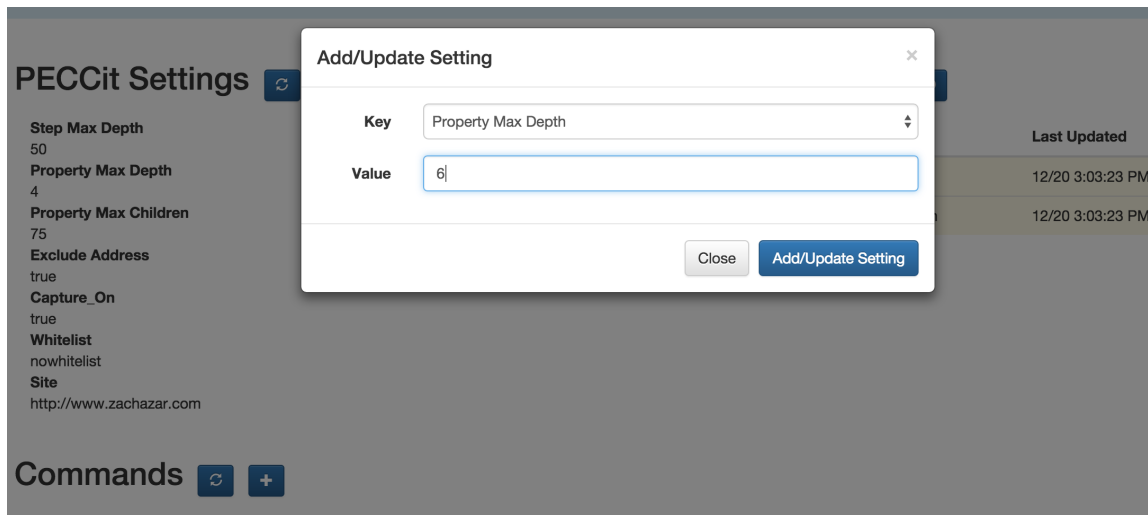
Command ID	From	To	Command	Received?	Issued At	Received At
1	Session Manager	ADS	Cancel	True	12/20 3:01:59 PM	12/20 3:01:59 PM
2	Session Manager	ADS	Cancel	True	12/20 3:02:36 PM	12/20 3:02:38 PM

© Zach Azar 2016

**Figure 3.4:** PECCit Session Manager: Launching the Inspector



**Figure 3.5:** PECCit Session Manager: Changing Settings



### 3.3.2 Changing Settings

Tracing all variables and values for a session can be quite expensive in terms of the time required and database space. Thus, it's best to adjust the settings before debugging a session. These settings, as listed in Section 3.5.1.3, control how many variables are saved and for which files to perform variable tracing.

Through the PECCit Session Manager, the user is able to change the active settings for the ADS. The current settings are listed in the table (see Figure 3.3) and they can be refreshed using the refresh icon. The user can click the + icon to bring up a dialog to change/add settings (see Figure 3.5). The ADS retrieves the settings at the beginning of each session so the desired settings for a session should be set prior to the execution.

### 3.3.3 System Status

As the ADS is a separate component from the web application, it can be helpful to see the status of the system (see Figure 3.3). Statuses include *Stopped*, *Running*, *Waiting for ADS*, and *Waiting for Connection*. Though the ADS has only been discussed so far, the PPM component is the PECCit Process Manager. It is the primary component that listens on the network for a possible session and passes off the connection to the ADS. With the System Status table, the user can manually refresh the status or set the application to automatically refresh the status table regularly.

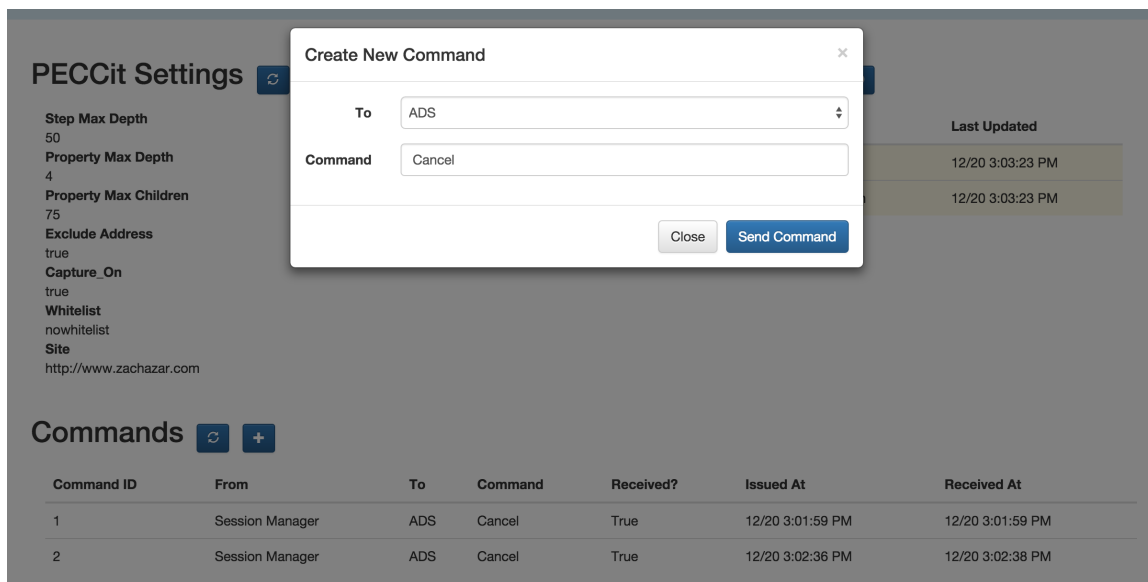
### 3.3.4 Sending Commands

The user can send commands in real-time to components in the system (see Figures 3.3 and 3.6). The only currently offered commands are *Cancel* and *Shutdown*. The *Cancel* command will cancel the current session that the ADS is tracing. The *Shutdown* command will cancel a session (if one is running), shutdown the ADS, and shutdown the PPM which quits the program. As the PECCit Session Manager and PECCit Inspector are separate components from the ADS, these web applications can still be used with the ADS shutdown. The table shows important information like when the command was sent and if/when the component received it.

## 3.4 PECCit Inspector

The PECCit Inspector looks like a modern debugger/IDE but is much more powerful (see Figure 3.7). With the Inspector, the user can debug a session by moving forward/back-

**Figure 3.6: PECCit Session Manager: Sending Commands**



ward through the trace, browse the source code with Execution Path Highlighting, inspect variable values and variable history, and watch previews of the page being built. As it is a web application, the user does not need to download any additional software to use the Inspector. Also since it's in the browser, multiple sessions can all be open at the same time. More information about the implementation of the PECCit Inspector is in Section 3.5. The following sections discuss the various features and tools of the PECCit Inspector.

### 3.4.1 Step Navigation

During a trace when a single line of code is executed, it is called a *step* in the PECCit system. The term *step* represents a line of code in a file being executed at a moment in time. Thus, the same line of code could belong to many steps (like a loop). The user is able

Figure 3.7: PECCit Inspector

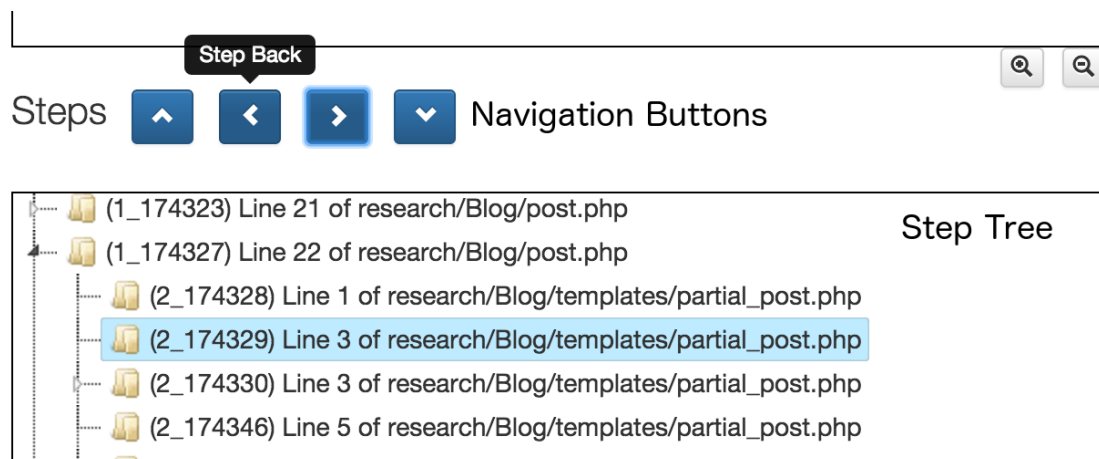
The screenshot displays the PECCit Inspector interface for Session 6. At the top, a teal header contains the text "PECCit Inspector for Session 6" and a "Back To Session Manager" link. Below this is a table with columns: SID, Query, Timestamp, Status, Steps, Time, Select Time, Insert Time, Messages, Inserts, DB Incr (MB), Settings, and Error Msg. The table shows a single entry for session 6 with a query to /research/Blog/ and various settings.

Below the table, a "Session" dropdown menu is set to "/research/Blog/ (12/20/15 4:05:29 PM)".

The main area is split into two panes. The left pane, titled "Source Code", shows the code for "research/Blog/templates/partial\_home.php". The code includes a PHP loop that iterates over \$allPosts, displaying the name and author of each post. The right pane, titled "Variables", shows the state of variables. It lists \$allPosts as an array of 3 posts. The first post has an author of "Bob" and content "Lorem ipsum dolor sit amet, consectetur adi". The second post has a name of "Post 2" and a slug of "p2". Other variables include \$pageName set to "Zach's Test Blog" and \$post as uninitialized.

At the bottom, a "Steps" section shows a list of execution steps. Step (2\_174217) is highlighted, corresponding to "Line 2 of research/Blog/templates/partial\_home.php".

**Figure 3.8:** PECCit Inspector: Step Tree

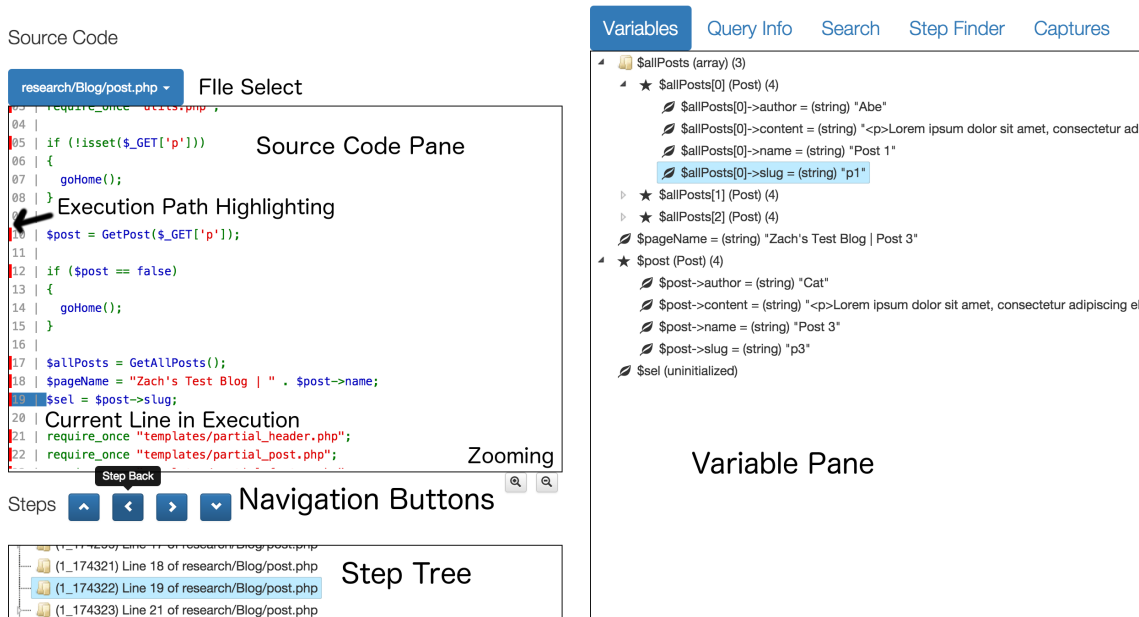


to navigate through these steps using the Step Tree and the Step Navigation Buttons (see Figures 3.8 and 3.9). When a user clicks a step in the Step Tree, the corresponding line of code is highlighted in the Source Code Pane. The user can click on any step in any order or they can use the Navigation Buttons. When right-clicked in the Step Tree, the user can search for all steps that occur at that same line of code (see Figure 3.22). This utilizes the Step Finder which is explained more in Section 3.4.6.

The user can click the navigational buttons to perform a Step Forward, Step Into, Step Back, and Step Out (see Figure 3.10). The Step Forward command will navigate to the next step at the current depth. The Step Into command will navigate to the next step that is deeper than the current step. For example, the user can use this command to follow the execution of a function call. The Step Out command will go back to the step that is one level higher than the current level (like stepping out of a function call). The final, and most



**Figure 3.9: PECCit Inspector: Labeled with Variable Pane Open**



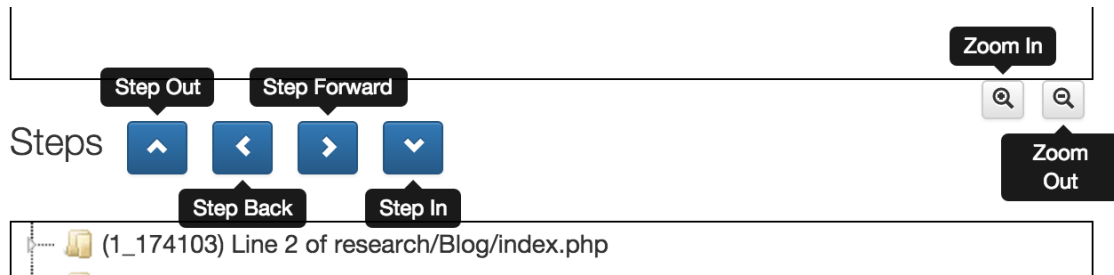
unique, navigational tool is the Step Back allowing the user to go back one step in time. This utility is not offered with modern debuggers.<sup>8</sup>

### 3.4.2 File Navigation and Execution Path Highlighting

The file list located above the Source Code Pane is a list of all of the files that were used during the execution of the session. The user can click a file in the list to load and view it (see Figure 3.11). The files are retrieved from the server using lazy loading so that they are only retrieved if they are needed when debugging in the Inspector. The source code is syntax highlighted and the user can click the magnifying buttons to zoom in and

<sup>8</sup>Technically, the Step Out button uses back-in-time functionality as well. Other debuggers use a Step Out which resumes forward execution until the end of the current function call. PECCit travels back-in-time to when the function was first called.

**Figure 3.10:** *PECCit Inspector: Navigation Buttons*



out. The blue highlighted line is the line that corresponds to the currently selected step in the Step Tree. This line represents the next step that will be taken in the execution (i.e. the blue line has not occurred yet). All lines of source code that are executed at some point during the session have red highlighting. This means that, without using the Step Tree and navigational tools, the user can quickly see which lines of code were executed. This feature is called Execution Path Highlighting and can be very useful when debugging. These features are shown in Figure 3.9.

### 3.4.3 Variable Pane, Variable Inspection, and Variable Differencing

PECCit offers multiple tools related to variables since often the most needed information when debugging are the variables and their values at certain times during the execution. When the user clicks on a step, the local variables that were present during that step in the execution are shown in the Variables Pane (see Figure 3.9). In this pane, variables are listed alphabetically. For scalar variables (boolean, integer, float, string), the variable is shown with a leaf icon and included is the variable's name, type, and value. Arrays are shown with a folder icon and include the name, type, and size. Objects are shown with a star icon and

**Figure 3.11:** PECCit Inspector: Source Code File Select

## Source Code

research/Blog/templates/partial\_home.php ▾

- research/Blog/index.php
- research/Blog/templates/partial\_footer.php
- research/Blog/templates/partial\_header.php
- research/Blog/templates/partial\_home.php
- research/Blog/templates/partial\_sidebar.php
- research/Blog/utils.php

```
07 |         <n4>written by <?php echo $post->author ?></h4>
08 |     </div>
09 |     <?php endforeach;?>
10 | </div>
11 | </div>
```

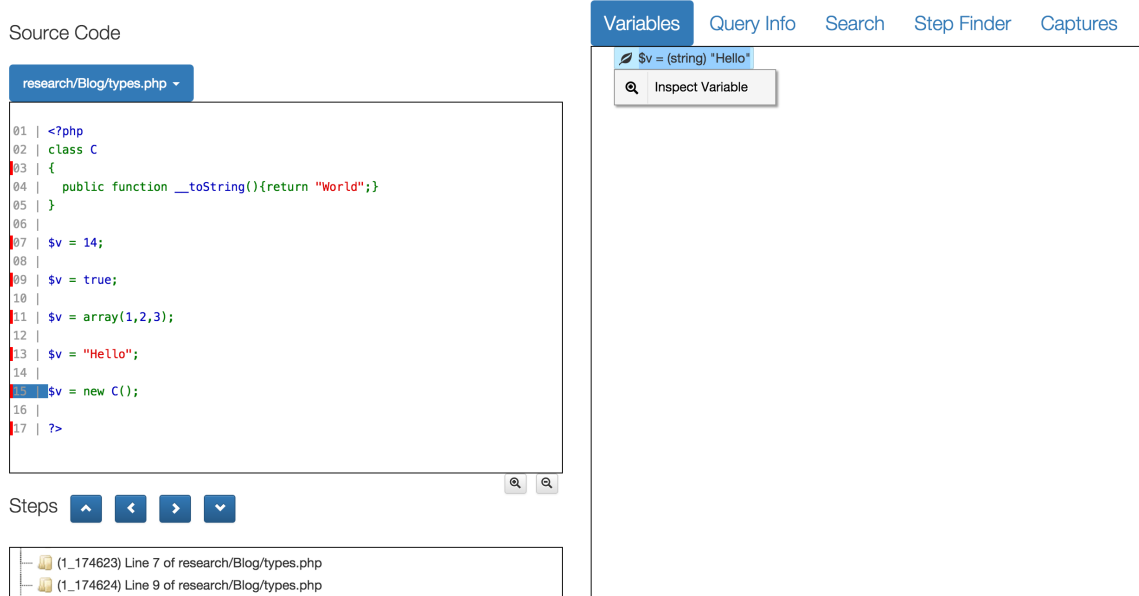
Steps ▲ ◀ ▶ ▼

list the name, classname, and number of properties. Users can click the arrays and objects in the Variable Pane to expand them and learn more about their contents and properties respectively. Examples of each of these types are shown in Figure 3.9.

A powerful tool that PECCit offers is Variable Inspection. The user can right-click a variable in the Variable Pane to bring up an option to “Inspect” the variable (see Figure 3.12). When clicked, PECCit will list every step when the variable’s value changed including when the variable was first initialized (see Figure 3.13). PHP is a dynamically typed programming language so variables can change types throughout their lifetime. Thus when Variable Inspection lists all of the steps where the variable changed, it also lists the variable’s type, value (if the type is scalar), classname (if it’s an object), facet (additional info like public, private, constant, etc.), and number of children/properties (if it’s an array or object). The steps also include a button which, when clicked, move the Step Tree to that step in the execution. For example, Figure 3.13 shows the locations where variable  $\$v$  changed. If the user clicks the second step button (where  $\$v$  is set to *14*), then the Inspector jumps to the location immediately after  $\$v$  is set to *14* as shown in Figure 3.14. This feature is similar to that of watchpoints described in Section 1.3. It instantly answers questions like “what values were assigned to  $x$  throughout the execution?” and “when was the last time  $x$  was changed?”.

Another useful tool that PECCit provides is Variable Differencing. Often, a developer just wants to quickly know what a line of code achieved during the execution without examining it in detail. With Variable Differencing, the user can select multiple steps in the Step Tree and the Variable Pane will show any variables that changed over those steps. This is useful for finding what kinds of side effects a deep function call might have. With

**Figure 3.12:** PECCit Inspector: Variable Inspect Tool



a standard debugger, the debugger only knows the variables at one moment in time and the user has to manually write down those values if they want to compare them. With PECCit, just ask if any differences occurred. For example in Figure 3.15, the user has highlighted three steps which essentially asks “show me every variable that changed from line 3 up to line 6.” In this case, the array was populated and the page name variable was changed.

### 3.4.4 Query Info Pane

The Query Info Pane shows the superglobal variables<sup>9</sup> that were available at the beginning of the execution. These include cookies, request parameters from GET/POST, and server info. This pane is purely informational and can be helpful when the user is curious

<sup>9</sup><http://php.net/manual/en/language.variables.superglobals.php>

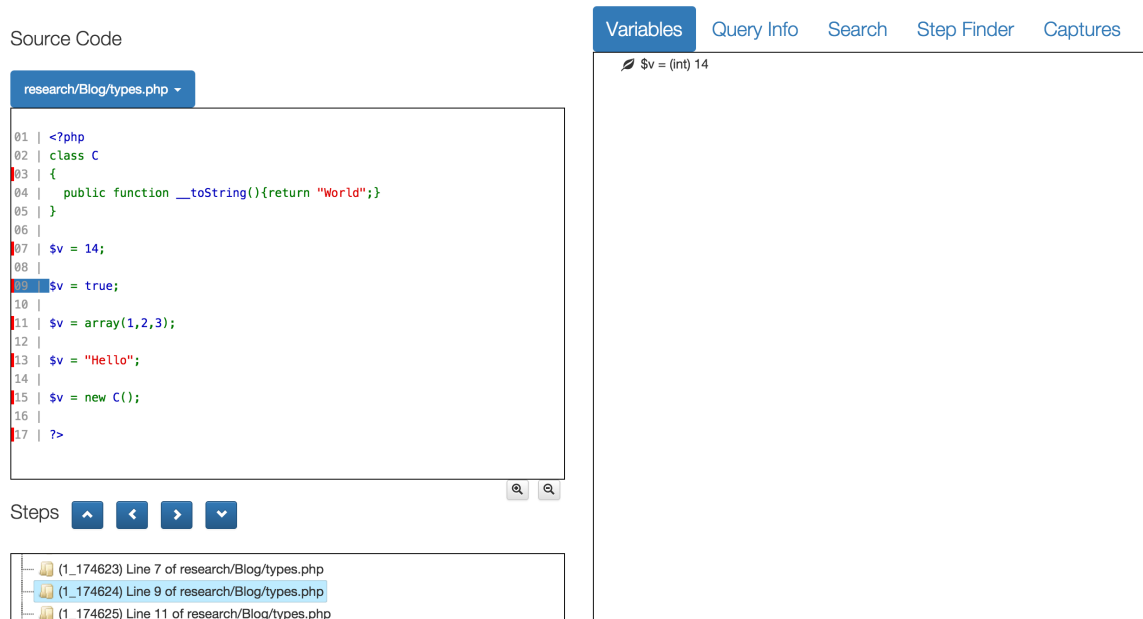
**Figure 3.13:** *PECCit Inspector: Inspection Results*

Variables Query Info Search Step Finder Captures

Variable: \$v

Location First Set	Value	Type	Classname	Facet	Number of Children/Properties
174622		uninitialized			
174624	14	int			
174625	1	bool			
174626		array			3
174627	Hello	string			
174628		object	C		0

**Figure 3.14:** PECCit Inspector: After \$v Has Been Set to 14



about how the request was received and what data was sent with the request (like cookie data). See Figure 3.16 for an example query where a GET parameter was specified. This is another feature that makes PECCit web development specific.

### 3.4.5 Search Tool

The Search Tool allows the user to search the variable names and values that were saved during the session. Users have the option of using wildcards in the search which will match items which *contain* the search term instead of perfectly matching it (for example, a search for 'data' would only match 'the\_database' if wildcards were used). Figure 3.17 demonstrates searching for a variable name and Figure 3.18 demonstrates a value search. When a variable is found, the user can use Variable Inspection by right-clicking the variable. For

**Figure 3.15:** PECCit Inspector: Example of Variable Differencing

Source Code

research/Blog/index.php

```
01 | <?php
02 | require_once "utils.php";
03 |
04 | $allPosts = GetAllPosts();
05 | $pageName = "Zach's Test Blog";
06 | require_once "templates/partial_header.php";
07 | require_once "templates/partial_home.php";
08 | require_once "templates/partial_footer.php";
09 | ?>
10 |
11 |
```

Steps

- (1\_174183) Line 2 of research/Blog/index.php
- (1\_174189) Line 4 of research/Blog/index.php
- (1\_174211) Line 5 of research/Blog/index.php
- (1\_174212) Line 6 of research/Blog/index.php

Variables

- \$allPosts (array) (3)
  - \$allPosts[0] (Post) (4)
  - \$allPosts[1] (Post) (4)
  - \$allPosts[2] (Post) (4)
- \$pageName = (string) "Zach's Test Blog"



**Figure 3.16: PECCit Inspector: Query Info**

SID	Query	Timestamp	Status	Steps	Time	Time	Time	Messages	Inserts	(MB)	Settings	Msg
7	/research/Blog/post.php?p=p3	12/20/15 4:38:01 PM	Done Collecting	97	00:00:00	00:00:00	00:00:00	531	2020	0.10938	Capture_On : true;Exclude Address : true;Property Max Children : 75;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : /;	

Session: /research/Blog/post.php?p=p3 (12/20/15 4:38:01 PM)

Source Code: research/Blog/post.php

```

01 | <?php
02 |
03 | require_once "utils.php";
04 |
05 | if (!isset($_GET['p']))
06 | {
07 |     goHome();
08 | }
09 |
10 | $post = GetPost($_GET['p']);
11 |
12 | if ($post == false)
13 | {
14 |     nonHome();

```

Variables Panel:

- \$GLOBALS (array) (8)
- \$\_COOKIE (array) (2)
- \$\_ENV (array) (0)
- \$\_FILES (array) (0)
- \$\_GET (array) (1)
  - \$\_GET['p'] = (string) "p3"
- \$\_POST (array) (0)
- \$\_REQUEST (array) (1)
- \$\_SERVER (array) (28)

example, Figure 3.18 shows the results of searching for the value *136*. The user inspects the variable using the Variable Inspector yielding results shown in Figure 3.19. Then, the user clicks the second step to see when (the immediate moment after) the variable was set to *136* (see Figure 3.19).

### 3.4.6 Step Finder Pane

The Step Finder Pane lets the user search for steps within a file. When the user specifies a file (and not a line number) and clicks “Find Steps”, the tool will find every time a step occurred within that file during the execution (see Figure 3.21). When a line is specified, the tool will return every time that specific line of code was executed. The user can then click the step button to navigate immediately to that moment during the execution. When the user right-clicks a step in the Step Tree, the Step Finder Pane is automatically opened,

**Figure 3.17:** PECCit Inspector: Search Tool for Variable Name

Variables   Query Info   **Search**   Step Finder   Captures














---

Settings   Display

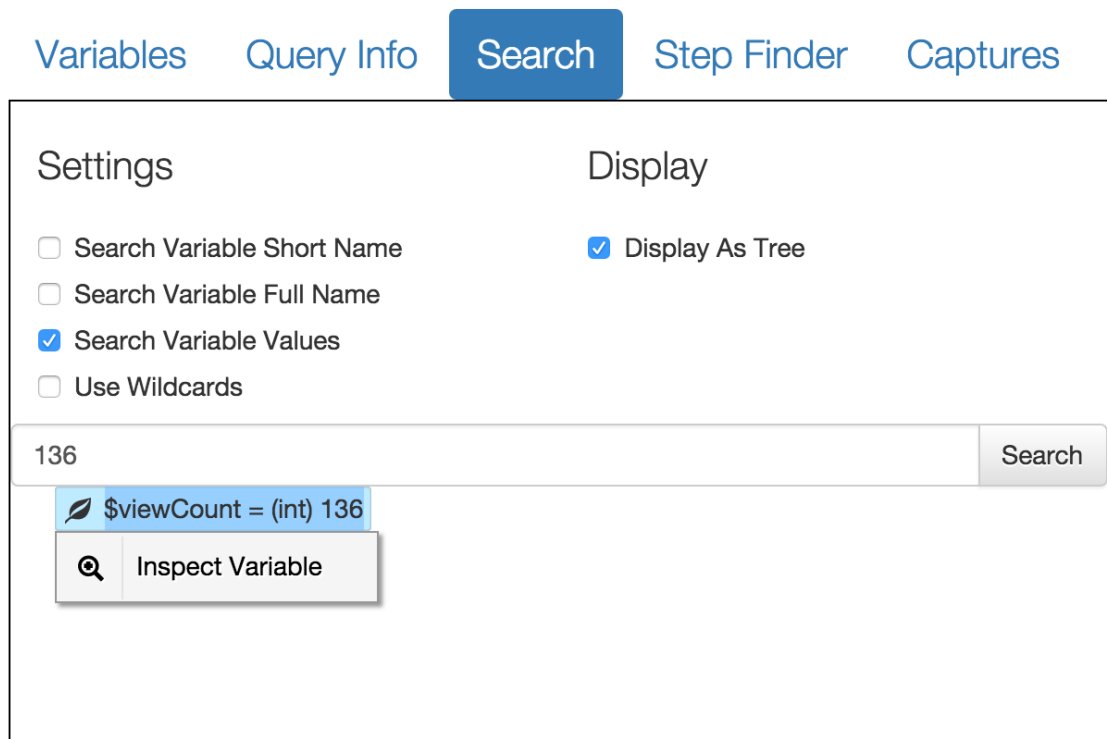
Search Variable Short Name       Display As Tree  
 Search Variable Full Name  
 Search Variable Values  
 Use Wildcards

---

post Search

-  \$allPosts (uninitialized)
-  \$allPosts (array) (3)
-  \$GLOBALS['\_POST'] (array) (0)
-  \$post (uninitialized)
-  \$post (Post) (4)
-  \$posts (uninitialized)
-  \$posts (array) (0)
-  \$posts (array) (1)
-  \$posts (array) (2)
-  \$posts (array) (3)
-  \$sidepost (uninitialized)
-  \$sidepost (Post) (4)
-  \$\_POST (array) (0)

**Figure 3.18:** PECCit Inspector: Search Tool for Variable Value



**Figure 3.19:** PECCit Inspector: Search Tool with Inspect Results

Variables Query Info Search Step Finder Captures

Variable: \$viewCount

Location First Set	Value	Type	Classname	Facet	Number of Children/Properties
174704		uninitialized			
174706	136	int			

**Figure 3.20:** PECCit Inspector: Jumping to Step From Search/Inspect Results

Source Code

research/Blog/templates/partial\_footer.php

```

1 | <?php
2 | $viewCount = GetViews();
3 | ?>
4 | <footer>
5 | <p>Page Views: <?php echo $viewCount; ?></p>
6 | <p>This is a test blog site written by Zach Azar</p>
7 | </footer>
8 | </body>
9 | </html>

```

Steps

- (1\_174703) Line 8 of research/Blog/index.php
- (2\_174704) Line 2 of research/Blog/templates/partial\_footer.php
- (2\_174706) Line 5 of research/Blog/templates/partial\_footer.php

Variables Query Info Search Step Finder Captures

- \$allPosts (array) (3)
- \$pageName = (string) "Zach's Test Blog"
- \$post (Post) (4)
- \$sel (null)
- \$sidepost (Post) (4)
- \$viewCount = (int) 136**

populated with the step's information, and searched (see an example in Figure 3.22). This tool can be helpful when the user wants to know every time a function was called, when a particular line of code was executed, or when execution first entered a file. In the example in Figure 3.22, the user can easily view every time a *Post* object was created. Execution Path Highlighting shows which lines of code were executed but the Step Finder allows the user to jump to that step immediately.

### 3.4.7 Capturing

Captures, presented in the Capture Pane, are previews of what a website looks like when it's being built during the execution. This is a novel feature provided by PECCit that is not offered by other debugging tools. For example, Figure 3.23 shows a capture of a test web page after the header was printed but the rest of the page hadn't been printed yet. Capturing must be specifically enabled in the settings and is only performed on whitelisted files.<sup>10</sup> When the user navigates through the steps with the Capture Pane open, they can watch as the page is being built line-by-line. The page content is presented in an `<iframe>` instead of a flat image so the user can still inspect and interact with the HTML, CSS, and JavaScript of the capture using their favorite browser tool like Chrome's DevTools<sup>11</sup>. For example, Figure 3.24 shows a capture that is being examined by the user using Chrome DevTools. Though it has a few shortcomings as discussed in Section 3.5.1.5, this is a very powerful tool.

---

<sup>10</sup>If the user wants to capture without variable tracing, just set the tracing settings conservatively by setting the Property Max Depth and Property Max Children settings to 0. See Section 3.5.1.3 for more detailed information about settings.

<sup>11</sup><https://developer.chrome.com/devtools>

**Figure 3.21:** PECCit Inspector: Step Finder

Variables   Query Info   Search   **Step Finder**   Captures

Find All Steps at Line Number

**Line Number (Optional)**

**File**

**Find Steps**

**Results:**

- Step: 174184
- Step: 174185
- Step: 174186
- Step: 174187

**Figure 3.22: PECCit Inspector: Step Finder from Step Tree**

Source Code

```

research/Blog/utlis.php
03 | class Post {
04 |     public $name;
05 |     public $slug;
06 |     public $author;
07 |     private $content;
08 |
09 |     public function __construct($n, $s, $a, $c)
10 |     {
11 |         $this->name = $n;
12 |         $this->slug = $s;
13 |         $this->author = $a;
14 |         $this->content = $c;
15 |     }
16 |
17 |     public function getContent()
18 |     {
19 |         return $this->content;
20 |     }
21 | }
    
```

Steps [Up] [Left] [Right] [Down]

Variables Query Info Search **Step Finder** Captures

Find All Steps at Line Number

Line Number (Optional)  
11

File  
research/Blog/utlis.php

Find Steps

Results:

- Step: 174639
- Step: 174645
- Step: 174651

Steps [Up] [Left] [Right] [Down]

(3\_174639) Line 11 of research/Blog/utlis.php  
List all Steps at this line Blog/utlis.php

**Figure 3.23: PECCit Inspector: Capture Showing Incomplete Web Page**

Source Code

```

research/Blog/index.php
01 | <?php
02 | require_once "utlis.php";
03 |
04 | $allPosts = GetAllPosts();
05 | $pageName = "Zach's Test Blog";
06 | require_once "templates/partial_header.php";
07 | require_once "templates/partial_home.php";
08 | require_once "templates/partial_footer.php";
09 | ?>
10 |
11 |
    
```

Steps [Up] [Left] [Right] [Down]

Variables Query Info Search Step Finder **Captures**

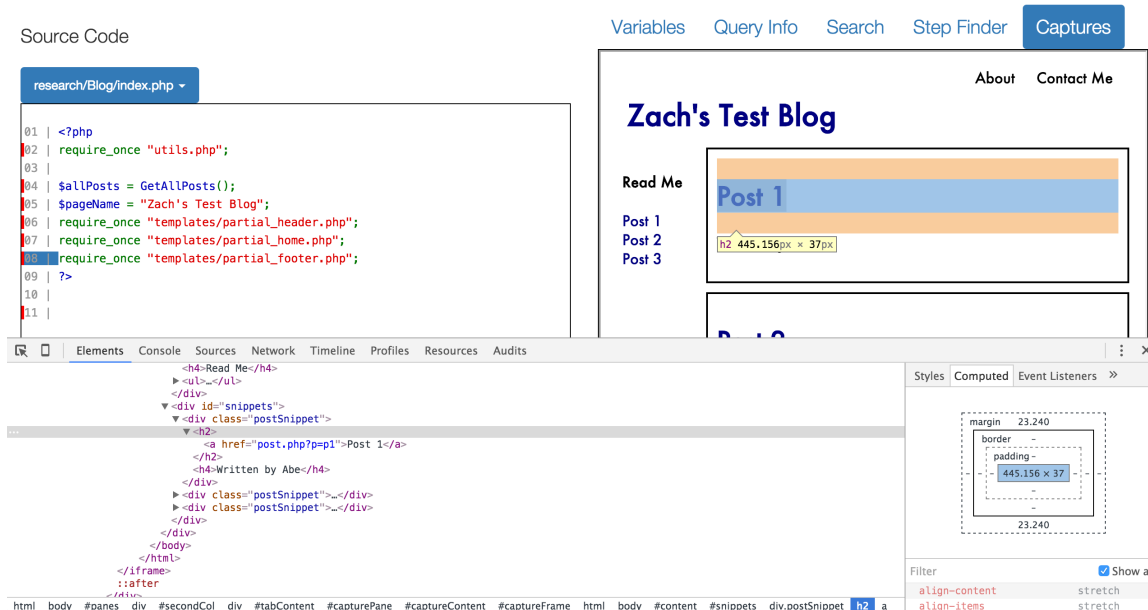
About Contact Me

## Zach's Test Blog

Steps [Up] [Left] [Right] [Down]

(1\_174189) Line 4 of research/Blog/index.php  
(1\_174211) Line 5 of research/Blog/index.php

**Figure 3.24:** PECCit Inspector: Using Chrome DevTools with Capture



### 3.5 Implementation

The PECCit system is comprised of various components and software systems that communicate with each other. As it is used to help developers debug and learn more about web applications which are inherently distributed systems, it itself is a distributed system with independent pieces. From a very high-level perspective, the PECCit system is composed of two main systems. The first is the Automated Debug Server (ADS). This system is responsible for recording everything that happens in a target framework during a web request. The second system is the PECCit Web Interface. This is responsible for displaying that information back to the user in a comprehensive and understandable way through the PECCit Session Manger and the PECCit Inspector components. These two systems are



shown in the high-level diagram of PECCit in Figure 3.1. This chapter describes how these systems work from a software level and the implementation behind PECCit's features.

### **3.5.1 Automated Debug Server**

The Automated Debug Server (ADS) is the backend component for the PECCit system. It is responsible for tracing the execution of a web request. The following subsections discuss how the ADS interacts with the execution using Xdebug, how the software was designed, how sessions are stored, how PECCit offers the novel feature of capturing, and how PECCit was designed to be language independent.

#### **3.5.1.1 Xdebug**

Xdebug<sup>12</sup> is an open-source extension for the Zend Engine<sup>13</sup>. It is written in C and was originally developed by Derick Rethans<sup>14</sup>. Xdebug is an incredibly powerful tool to use when developing PHP websites. It has the ability to better visualize and dump variables, collect function and stack traces, and even analyze code via code coverage and profiling. The most important feature that it has, for PECCit, is the ability to debug PHP scripts remotely.

Xdebug's remote debugging functionality is feature rich and is used by many popular IDEs. Xdebug lists some of these IDEs<sup>15</sup> including Eclipse, Emacs, Komodo, Notepad++, NetBeans, etc. Once Xdebug is compiled and installed, PHP is configured to use Xdebug, and Xdebug's remote debugging is enabled, a remote debugger is able to use Xdebug to

---

<sup>12</sup><http://xdebug.org/>

<sup>13</sup><https://www.zend.com/en/community/php>

<sup>14</sup><http://derickrethans.nl/>

<sup>15</sup><http://xdebug.org/docs/remote>

step through the execution of a PHP script and perform the basic functionality of a debugger. Xdebug and the remote interface use an open-source protocol to communicate called DBGp<sup>16</sup>.

Using DBGp, the remote debugger is able to perform multiple debugging actions including the following:

- Set and get breakpoints
- Step into, step over, step out, stop, and other commands that affect the continuation of the debugging session
- Retrieve the call stack and stack depth
- Retrieve the context including variables and their values
- Evaluate and execute pieces of code and expressions

When configured and the user provides a remote host address and remote port, Xdebug communicates with the remote IDE using the DBGp protocol over TCP on the network. When a request comes in to the server to execute a PHP script, Xdebug will check the request to see if the user wants to debug the request. A user specifies if they want to debug a request by including information either in the URL GET parameters, in the POST data, or as a cookie. Conveniently, there are extensions for the popular browsers which will include this Xdebug metadata automatically. When developing PECCit, the Chrome extension *Xdebug Helper*<sup>17</sup> was used. When a request for a page is made and the request is sent with

---

<sup>16</sup><http://xdebug.org/docs-dbgp.php>

<sup>17</sup><https://github.com/mac-cain13/xdebug-helper-for-chrome>

the Xdebug metadata, Xdebug will ask the remote host if it would like to remotely debug the request.

### 3.5.1.2 ADS Design and Workflow

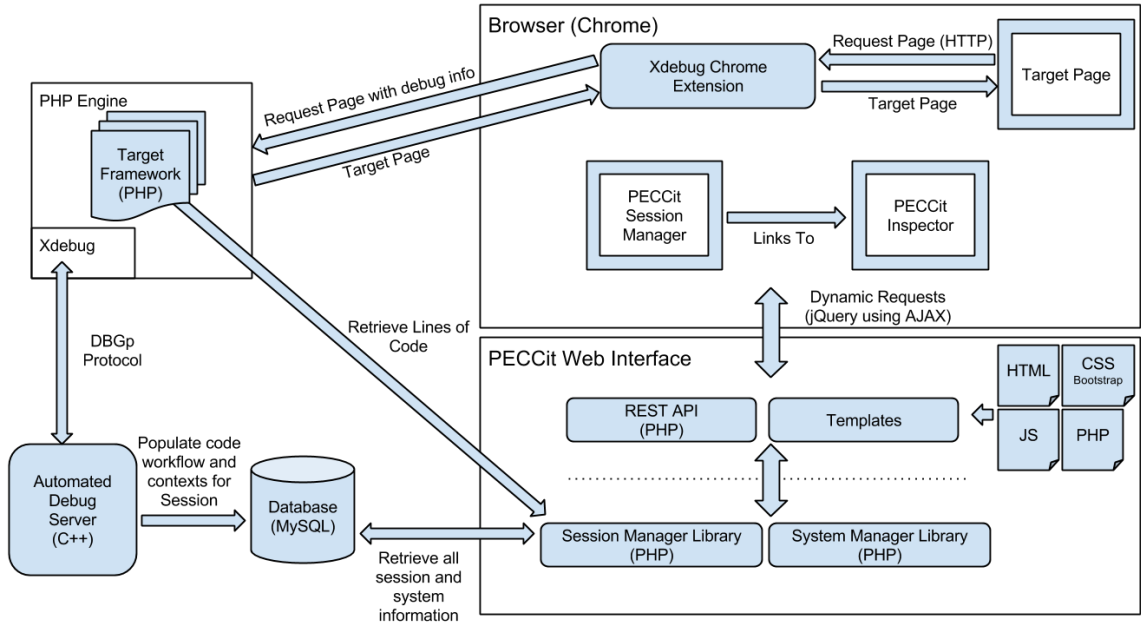
The ADS acts as an automated, remote debugger that communicates with Xdebug. Figure 3.25 demonstrates this interaction. In the figure, the top right outer box is the user's browser.<sup>18</sup> The user first requests the target page (TP) over HTTP (like `http://example.com/index.php`). The Xdebug Chrome Extension adds Xdebug information to the request before it is sent out. The request is sent to the server where the target framework (TF) exists. The TF could be a common framework like WordPress or Drupal, or it could be a custom PHP site. When the request comes in, it is handled by the PHP engine. Before executing, Xdebug sees the extra debug information in the request and stops the execution. It contacts the remote host to see if it would like to perform remote debugging. In the case of the PECCit system, the remote host is the ADS.

The ADS proceeds to interact with Xdebug during the execution. It regularly asks Xdebug, using the DBGp protocol, to perform debugging actions like step into/over the next statement, retrieve the stack depth, retrieve the variables and values, etc. During the execution, all of the data retrieved is saved in the database as a *session*. Once the execution is complete, the page that the PHP engine built is returned back to the user's browser (as shown in Figure 3.25) and the ADS finalizes the session in the database. The ADS then waits on a socket (hence why *server* is in the name) for another session to connect.

---

<sup>18</sup>Chrome was used for developing and testing PECCit.

**Figure 3.25: PECCit Workflow Diagram**



Specifically, the PECCit Process Manager (PPM) waits on the socket and when a new connection is made, the request is handed off to the ADS.

The ADS was written in C++ and was designed and tested on various Unix machines (Mac OS X and CentOS).<sup>19</sup> The ADS executable is started through the command line. The program was designed to use MySQL<sup>20</sup> for the database (though this could be abstracted in future work). The installation of MySQL creates a `mysql.h` file which includes definitions of various necessary classes and functions that PECCit uses. Also, the ADS uses the MySQL Connector/C<sup>21</sup> library to interact with the database.

<sup>19</sup>As the ADS is a prototype used in research, it has not been tested for robustness on various operating systems and compiler versions (including on Windows machines).

<sup>20</sup><https://www.mysql.com/>

<sup>21</sup><http://dev.mysql.com/doc/connector-c/en/index.html>

Another dependency that PECCit uses is an XML parsing and traversal library called Rapid XML<sup>22</sup>. This lightweight library is written in C++, is licensed under the MIT license<sup>23</sup>, and was created by Marcin Kalicinski. Since Rapid XML is a library, it does not require installation. Rapid XML is included in the PECCit software project unaltered. Rapid XML allows for very fast traversal of XML which is crucial for parsing the responses sent from Xdebug.

### 3.5.1.3 PECCit Settings

Tracing can be quite expensive in terms of execution overhead, database storage, and memory usage. Though it's possible to fully trace a session such that every line of code is traced and all variables are saved, it is often too costly and the extra data is not needed during debugging. To make PECCit more practical, various settings can be used to reduce the amount of tracing that the ADS performs during an execution. These settings are listed below:

#### Whitelist

The *Whitelist* is the most important setting in the PECCit system. The ADS will only save variable data and perform capturing when the execution is currently in a file on the whitelist. The whitelist is a list of terms (semicolon separated) that are matched against file paths to see if the file path *contains* the term. For example if the whitelist is “index;utils”, then “/index.php” will be traced, all files under “/utils/” will be traced, but “/post.php” will not be traced. Thus, it can be used to specify

---

<sup>22</sup><http://rapidxml.sourceforge.net/>

<sup>23</sup><http://rapidxml.sourceforge.net/license.txt>

individual files or entire subdomains. For files not on the whitelist, the execution path will still be saved but the variable data and captures will not be. The *Whitelist* setting is important because saving variable data and captures is expensive and should only be performed on files/folders where the developer suspects a bug. The default is “/” which specifies that all files are on the whitelist.

### **Step Max Depth**

As more and more functions are called, the call stack gets deeper accordingly. By default, PECCit will perform a *Step Into* into each of these function calls. This can sometimes get expensive however (for example, a deep recursive call). Thus, the system will start to perform a *Step Over* once the call stack depth reaches the *Step Max Depth* setting. It will resume using *Step Into* commands once the current stack depth is below the *Step Max Depth* setting. The default is 50.

### **Property Max Depth**

Properties (variables) can potentially own additional variables (for example, an array or object). These owned properties could own their own properties and this ownership chain can go arbitrarily deep (or even infinite loop if the object refers to itself or children refer to their parents). The ADS will only ask for more information about property children up to the depth set by the *Property Max Depth* setting. The default is 4.

### **Property Max Children**

As mentioned, properties can own other properties (like an object or array). The ADS will automatically ask for more information for each of these children. Thus,

a property with lots of children could slow down the trace considerably. To increase performance, the ADS will only ask for more information from the children up to the number set by the *Property Max Children* setting. The default is 75.<sup>24</sup>

### **Check Commands Frequency**

Internally, the ADS is looping and sending hundreds of messages to Xdebug to complete the trace. If the ADS checked for new commands from the user after every message (see Section 3.3.4 about sending commands to the ADS), performance would drop dramatically. Thus, it checks after a certain number of cycles as set by the *Check Commands Frequency* setting. If this number is large, it checks less often resulting in a performance boost. However the longer it waits to check, the longer a command is delayed before it's received by the ADS. The default is 100 which results in a fairly instant response.

### **Exclude Address**

This is a very technical setting for how the database should compare variables. In the database, each variable is stored with an address field which is its location in memory during the execution. If this setting is set to *true*, then PECCit does not use the memory address field when comparing two variables to see if they are the same. When set to *false*, the addresses of the variables are used. The address field can help to distinguish between two variables that are seemingly identical (same name, type, etc.). Further testing with this setting could yield performance improvements. The default is *true* and the setting should not be changed without testing.

---

<sup>24</sup>Xdebug itself must actually limit the number of children that can be asked for. If this number is set quite large, Xdebug slows down dramatically. Thus, a very large *Property Max Children* should not be used.

## Capture On

The *Capture On* setting enables/disables capturing when set to *true/false* respectively. Capturing, as shown in Section 3.4.7, can be used to preview a site as it's being built line-by-line. Capturing can dramatically reduce performance however and should only be used when it's needed. The default is *true* which automatically enables capturing.

## Site

The *Site* setting is the name of the top-level domain of the website. For example, "http://www.zachazar.com". This setting is used when displaying captures in the PECCit Inspector. Specifically, the *iframe* that houses the capture is given a *<base>* tag with the *Site* setting so that links are displayed correctly in the *iframe*.

### 3.5.1.4 Database and Session Storage

The database is an important component of the PECCit system. All session and system information is stored in the PECCit database. The database used is MySQL<sup>25</sup> since it's popular, fast, and open-source. The ADS communicates with the database to save session information and retrieve settings and commands. The web interface uses the database to retrieve the session info and update commands and settings.

As discussed in Section 3.4.1, a *session* is primarily composed of a collection of *steps*. Each step represents a line of execution that the program performed. Thus, a step is simply a line number and a reference to a file. The order in which these steps occur signifies the order of the execution statements during the program. Throughout the execution of the

---

<sup>25</sup><https://www.mysql.com/>



program, there are various *variables* that go in and out of scope. As variables can take on different values (and in PHP, these variables can take on different types), the PECCit system stores these variable values as *contexts*. Then for each step, the particular context of a variable is saved in a *closure* connecting the step to all of the variable values that were in scope during that step. PECCit uses this somewhat complex schema to reduce repetition but still store all necessary data and references.

The database stores other information that is also important for sessions and system operations. Each session has analytics data that saves information about timing, number of messages, errors, etc. The database also stores captures which are linked to the steps to provide web page previews for the user. For system operations, the database stores statuses for the components, commands sent from the components, and PECCit system settings. This schema is crucial for PECCit operations, though experimentation with other architectures and technologies could improve performance.

### **3.5.1.5 Capturing**

Capturing is a novel tool that PECCit provides web developers. It allows developers the ability to watch as the web page is being built line-by-line (after the execution). The core technology that enables PECCit to allow capturing is PHP's Output Buffering<sup>26</sup>. When Output Buffering is enabled (which it is, by default, on production servers), output from PHP that is sent to the browser is first stored in an output buffer. Once this output buffer fills up, the execution ends, or the user specifically flushes it, the contents of the output buffer are flushed and sent to the browser. If the PHP script is still creating more content for the

---

<sup>26</sup><http://php.net/manual/en/book.outcontrol.php>

page, new content goes into the now-empty output buffer and the process continues. This mechanism improves performance as it is more efficient to send medium sized messages to the browser rather than a large number of small ones.

Output buffering can also be controlled by the user. A developer can programmatically control when the buffer is flushed, how much is saved, and they can even retrieve the contents of the buffer without sending it. This feature can be used to change something about the page after it's built or gain information about the complete page like total page size or time to build. Output buffering can also be helpful when using template files to collect their output and use the output within other template files.

For PECCit, the output buffer presents the opportunity to save what the page looks like after every step. To do this, PECCit inserts code into the beginning of the execution (using Xdebug) which enables output buffering with a very large buffer.<sup>27</sup> Once enabled, the output buffer slowly fills with the page that would normally be sent to the browser. Then during execution when on a whitelisted page, the ADS inserts code using Xdebug to retrieve everything in the output buffer (and not flush it). This data is called a *capture* in the PECCit system and is stored in the database. When the execution finishes, the output buffer is flushed automatically and sent to the browser.

One drawback to capturing is that it has a direct impact on the execution that it is debugging. As mentioned, capturing inserts code into the running execution. This could have undefined side effects on the execution if the framework being debugged strongly relies on output buffering being disabled or uses output buffering itself. When testing PECCit though, having capturing enabled did not appear to negatively impact WordPress

---

<sup>27</sup>See <http://php.net/manual/en/function.ob-start.php> for more details on enabling output buffering.

nor Drupal frameworks which both use output buffering to buffer template files. Further testing on the impacts of output buffering could be beneficial in future work.

### 3.5.1.6 Language Independence

PECCit is designed to be language independent. The system, unlike most omniscient and back-in-time debuggers, does not use code injection nor virtual machine control (unless capturing is enabled). Instead, it gains all of its information about the execution from a debugging engine offering remote debugging. PECCit is designed to interact with the debugging engine using the DBGp<sup>28</sup> protocol.

Thus, PECCit could be used with other languages/systems if they offer remote debugging using DBGp. ActiveState<sup>29</sup>, a company which sells a powerful IDE called Komodo<sup>30</sup>, offers remote debugging software that use DBGp for debugging Perl, Python, Ruby, and Tcl.<sup>31</sup> These tools can be used by other IDEs, like PECCit, to remotely debug an execution in one of these languages. For example, Vdebug<sup>32</sup> is a multi-language debugger that relies on remote, DBGp debugging and the authors encourage the use of ActiveState's components.<sup>33</sup>

Though in theory PECCit could debug any language with a DBGp compliant debugger, the system has not yet been tested to do so. There may be known and unknown complications within the ADS and PECCit Web Interface that need to be addressed. For example,

---

<sup>28</sup><http://xdebug.org/docs-dbgp.php>

<sup>29</sup><http://www.activestate.com/>

<sup>30</sup><http://www.activestate.com/developer-tools/komodo/komodo-ide>

<sup>31</sup>Their latest versions of these remote debugging components can be found at <http://downloads.activestate.com/Komodo/releases/9.3.2/remotedebugging/>

<sup>32</sup><https://github.com/joonty/vdebug>

<sup>33</sup>See Vdebug's documentation at <https://github.com/joonty/vdebug/blob/master/doc/Vdebug.txt> .

the ADS relies on the debugger offering certain features like retrieving the call stack and variable values. If the DBGp compliant debugger didn't offer those functions, PECCit's tracing might be corrupted. Also, certain pieces of the DBGp messages are optional (like providing a classname when retrieving an object) which could interfere with PECCit's tracing ability. The PECCit Web Interface also hasn't been tested with other languages and issues may occur. For example, one known issue is the PECCit Inspector always uses syntax highlighting for the PHP language and does not check for the language of the source code. Through testing and further improvements (see Section 5.1.3), PECCit could potentially be used with many other languages besides PHP.

Another known limitation to language independence is PECCit's capturing functionality. Currently, as described in Section 3.5.1.5, PECCit retrieves captures using output buffering, a function specific to PHP. However, a form of output buffering is offered by other languages as well. Ruby on Rails offers *Streaming*<sup>34</sup>. ASP.NET offers *Response Buffers*<sup>35</sup>. Perl offers buffering through its *IO:Handle*<sup>36</sup>. These technologies could be utilized to offer capturing with other languages as well, though further testing and development would be needed (see Section 5.1.3).

### 3.5.2 PECCit Web Interface

The PECCit Web Interface provides the two web applications that can be used to access the PECCit system: the PECCit Session Manager and the PECCit Inspector. When the user requests one of these applications, it is built dynamically using a combination of template

---

<sup>34</sup><http://api.rubyonrails.org/classes/ActionController/Streaming.html>

<sup>35</sup>[https://msdn.microsoft.com/en-us/library/ms526001\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/ms526001(v=vs.90).aspx)

<sup>36</sup><http://perldoc.perl.org/IO/Handle.html>

files (HTML, CSS, JavaScript, etc) and information from the database (as shown in Figure 3.25). These applications are capable of presenting and altering data in the database and managing sessions. The following subsections describe the implementation of these applications and how the web Interface constructs them.

### 3.5.2.1 Handling the Data

There is a lot of data contained in the PECCit system with large variable traces, captures, source code, analytics, commands, etc. PECCit utilizes two classes in the PECCit Web Interface which manage this data. The first, and most important, is the *SessionManager* class. This PHP class is responsible for accessing all session information in the database. It is able to retrieve a list of sessions, all of their variables/values, captures, analytics, settings, etc. It can also read, syntax highlight, and return source code files. It is the primary utility class to access the database and debugging information.

The second necessary class that the Web Interface uses is the *SystemManager* class. It is capable of reading the system statuses of the different PECCit components in the database and can send messages to the different components through the database. The *SessionManager* and *SystemManager* classes are shown in Figure 3.25 along with their interactions with the Target Framework (for source code) and with the database (for session and system information).

Using these two classes, the PECCit Web Interface offers a REST API<sup>37</sup> to handle the PECCit system data. The API is capable of returning session information, system

---

<sup>37</sup>Representational State Transfer (REST) is a common protocol that client-server architectures often use to request, alter, and transfer data. More information can be found at [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) or [https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer).

commands and statuses, captures, source code, etc. The data returned is often in JSON format or raw HTML to be inserted into a web page using JavaScript. The REST API also allows for modifications to the system like changing settings, sending commands, and deleting sessions.

### 3.5.2.2 PECCit Session Manager Implementation

As discussed in Section 3.3, the PECCit Session Manager is a web application which manages and controls the entire PECCit system. It is built by the Web Interface using PHP and template files. When built, the most recent session/system data is built into the page using the *SessionManager* and *SystemManager* classes.

The application utilizes JavaScript (specifically, jQuery<sup>38</sup>) to respond to user interactions. The JavaScript code uses AJAX<sup>39</sup> to query the REST API to adjust the page dynamically without having to reload the page. For example when the user clicks to refresh the session list (see Figure 3.2), a JavaScript function is fired which makes an AJAX request to the REST API on the server asking for the list of sessions. When it receives the list, a callback function is executed which replaces the list being displayed on the user's browser with the latest list received from the server. It uses AJAX to refresh and delete sessions, refresh and change settings, refresh and send system commands, and refresh the system status list.

---

<sup>38</sup>jQuery is a JavaScript library that improves upon JavaScript functions including handling AJAX calls and traversing/modifying the DOM. See <https://jquery.com/> for more information.

<sup>39</sup>AJAX is a strategy for querying the server for data without the need to reload the page. See <https://developer.mozilla.org/en-US/docs/AJAX>

The PECCit Session Manager (as well as the PECCit Inspector) primarily uses Bootstrap<sup>40</sup> for appearance and web page layout. Bootstrap is an open-source, frontend framework that enables a developer to quickly build responsive web applications that look great without much additional styling. In the PECCit Session Manager, Bootstrap supplies the basic styling (layout, fonts, tables, etc.) and a few JavaScript/jQuery plugins (the modals used for creating settings and commands). Of course, PECCit also has custom CSS to adjust styling but Bootstrap greatly assisted in getting PECCit up and running quickly by providing lots of features out of the box.

### 3.5.2.3 PECCit Inspector Implementation

The PECCit Inspector is a much more complicated component than the Session Manager. As discussed in Section 3.4, the PECCit Inspector is capable of browsing source code and variables, searching for variables, and displaying captures for a session. When the page is first requested by the user, not everything is built into the page as there is a lot of information involved in a session. Instead, only the basic layout and minor information is returned with the page.

The PECCit Inspector uses AJAX and lazy-loading to improve performance and usability. Once the page is returned to the user, an AJAX request is sent to the REST API to retrieve the *steps* for the session. This takes time (as there are typically thousands of steps in a single request) so retrieving them using an AJAX request allows the user to explore the Inspector while they load. Then additional information is only retrieved when the user asks for it (lazy-loading). For example when a user clicks a step in the Step Tree (see Figure

---

<sup>40</sup><http://getbootstrap.com/>

3.8), the source code for that file, the variables for that step, and the capture are retrieved dynamically using multiple AJAX requests to the REST API. This greatly improves performance as the user might not need all information for all steps and all source code files when they're debugging. Once something is retrieved (like the source code for a file or a list of variables), then it is saved in the browser so it won't need to be retrieved again. The other tools like Step Finder and Search also utilize AJAX and REST. This strategy allows the PECCit Inspector to have reasonable load times while still allowing access to a vast amount of information (as the user needs it) without needing to reload the page.

The Step Tree and Variable Pane (see Figure 3.9) are powered by a JavaScript library called jsTree<sup>41</sup>. jsTree is open-source under the MIT License<sup>42</sup> and is capable of creating and displaying interactive trees to the user. The user can expand tree items, drag and drop, search, etc. using the library. On the PECCit Inspector, jsTree is used to display the steps and variables to the user. This enables the user to see the steps and variables in an expandable and easy-to-read format. On creation, the tree is customized in the PECCit Inspector for certain icons and clicks. The library itself is unaltered however.

Similarly to the PECCit Session Manager, Bootstrap is used for basic styling and layout. In addition to layout and font, it provides the input groups used by the Step Finder and Search, and the tooltips used by the step navigation buttons (see Figure 3.10). Most importantly, it provides the tabbing functionality used by the various panes.

Captures, similarly to variables, are retrieved from the REST API using AJAX when the user clicks on a step. The contents of the capture (HTML, CSS, JavaScript, etc.) are

---

<sup>41</sup><https://www.jstree.com/>

<sup>42</sup><https://opensource.org/licenses/MIT>



displayed to the user using an `<iframe>`<sup>43</sup>. As discussed in Section 3.4.7, this iframe acts like a small browser so the user can interact with the contents of the capture like it's an actual web page (and debug it using browser tools like Chrome's DevTools<sup>44</sup>).

---

<sup>43</sup><https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe>

<sup>44</sup><https://developer.chrome.com/devtools>

# Chapter 4

## Evaluation and Analysis

Evaluation of a software tool like PECCit can be difficult without a large, human-based empirical study. Often, tools will do performance testing ([LGN08, LHS97, BM14]) to show that their tool doesn't require much overhead (at least compared to similar tools) but this doesn't necessarily support that the tool is useful. Since PECCit doesn't aim for performance benchmarks, performance testing wouldn't be an effective evaluation. Other tools (in addition to performance testing) will perform user studies with a handful of developers ([KM08, MBP11a, BBKE13]). Though encouraging, these studies typically have far too few participants to support a definitive claim of value presented by the tool.

Other researchers used their tool on real-world problems to demonstrate its effectiveness. For example, one group used their tool to find bugs in real-world web applications [ADTP10]. Another tool was used to find a bug in the popular Firefox browser [KFH13]. Another tool was shown to help a developer answer possible conjectures about two Java frameworks [GDJ02]. These case studies demonstrate effectiveness, though admittedly

they do not prove it. As PECCit was designed to offer features not previously offered by similar tools (omniscient debugging for backend web development), the case study strategy was used to demonstrate its effectiveness.

The case studies chosen were questions/topics/bugs brought up by real people from support forums. Four of the case studies use the WordPress framework<sup>1</sup> from the corresponding support forum<sup>2</sup>. One case study uses the Drupal framework<sup>3</sup> from its support forum<sup>4</sup>. WordPress and Drupal are popular Content Management Systems (CMS) with users ranging from no coding experience to advanced developers. The two frameworks were both chosen because there are thousands of posts for support for each system, they are complex frameworks with many working pieces, and the backend of both WordPress and Drupal are written in PHP. The goal of these case studies is to show how a developer new to WordPress or Drupal could use PECCit to solve potentially complex problems. The case studies do not necessarily solve the problems but often lead the developer into a direction to solve them.

The case studies were not picked at random, but were selected based on their presumed likelihood of demonstrating the various features of PECCit. PECCit traces the backend of a web framework as it is building the content of a page. Thus, forum questions involving front end appearance (i.e. “the font is too small”) or JavaScript interaction (i.e. “the button doesn’t work”) were ignored as they are not (typically) created by backend bugs but rather CSS and JavaScript (though as you will see in Case Study 5, the ‘bug’ was actually in CSS). Other forum posts involving separate entities like cacheing, security, e-commerce,

---

<sup>1</sup><https://wordpress.org/>

<sup>2</sup><https://wordpress.org/support/>

<sup>3</sup><https://www.drupal.org/>

<sup>4</sup><https://www.drupal.org/project/issues/>

and social media connectors were also ignored since reproducing the post's reported bugs would be nearly impossible without the user's credentials to the third parties often used by these entities. A number of posts do not report buggy activity but rather ask about plugin features (or request them). These posts were also ignored as only posts reporting buggy activity most likely caused by the backend framework were considered.

The researcher in these case studies was moderately experienced with web development but was only slightly experienced with WordPress and Drupal. The researcher had previously developed with HTML, CSS, PHP, and JavaScript. They were familiar with the basic functionality of WordPress and Drupal, but were inexperienced with more advanced topics like plugin/theme development, AJAX requests, and how WordPress and Drupal build pages. The user was familiar with how to debug with PECCit including how to use the settings properly.

## **4.1 Case Study 1: Non-Admin Can Upgrade Database**

### **4.1.1 Background**

The WordPress community regularly releases major and minor updates to the WordPress core framework and plugins to fix bugs, to patch issues, etc. To update WordPress, the site administrator can either:

- Use the *wp-admin* web interface to perform the update through the browser
- Setup the site to perform automatic updates without the site administrator needing to interfere

- Update manually by downloading the latest version on the server and replacing the necessary files

For each of the techniques after the files have been updated, the database needs to be updated. The update often needs to change the structure/schema of the database or its content. While the database update is in progress, the site might have strange behavior as users might retrieve an old or partial copy during the update. It is best to perform updates at low-usage times and on a test site first.

### 4.1.2 Problem

Typically after updating a plugin, the site administrator visits any link under the */wp-admin/* subfolder and the WordPress framework redirects the administrator to a page which encourages the administrator to perform a database update (WordPress will perform the update for you). In this ticket<sup>5</sup> (see Figure 4.1), the issuer noticed that if anyone (even anonymous users) go to */wp-admin/*, the WordPress core will ask if they want to update. This includes visitors who are not logged in as an administrator.

This could result in various problems. Maybe the administrator has purposefully not performed the update because:

- They think it could damage the site.
- The site/server is experiencing heavy traffic and they worry it could result in down time.

---

<sup>5</sup><https://core.trac.wordpress.org/ticket/34200>

- The administrator is performing a database backup (which is recommended) in case the update breaks something.
- Probably most dangerously, maybe the site updated automatically but the administrator hasn't logged in yet to update the database (and perform a backup of the database first).

Since an anonymous user can update the database without an administrator being aware, they could inadvertently/maliciously affect the website. By updating the database, they could cause the site to have downtime during high peak usage, it could cause errors to the appearance, or even potentially expose security concerns.

### 4.1.3 Setup


The issuer of the bug said that they noticed the update request when they upgraded from WordPress version 4.30 to 4.31 (though this apparently occurs with most, if not all, updates). To prepare for debugging, a fresh install of WordPress using version 4.30 was installed on the development server. The site was then upgraded the site to 4.31. As the development server used is not configured to allow WordPress to update itself, the manual method was used for the update. As an anonymous (non-logged in) user, the */wp-admin/* subfolder was visited in the testing browser and confirmed that it does ask to update the database without needing to login (see Figure 4.2). In this case study, assume the developer is not experienced with WordPress and does not know how to fix this problem or even learn more about it.

**Figure 4.1: Case Study 1: Forum Post**

[#34200](#) [new defect \(bug\)](#) Opened [91 minutes ago](#)  
Last modified [75 minutes ago](#)

---

**Prompted to upgrade database when visiting /wp-admin/ and logged out**

Reported by:	 <a href="#">morganestes</a>	Owned by:	
Milestone:	<a href="#">Awaiting Review</a>	Priority:	<a href="#">normal</a>
Severity:	<a href="#">normal</a>	Version:	<a href="#">4.3.1</a>
Component:	<a href="#">Database</a>	Keywords:	
Focuses:	<a href="#">administration</a>		

---

Description


When accessing /wp-admin/ while signed out on a site that was autoupdated from 4.3.0 to 4.3.1, I was prompted to update the database before continuing, even though I wasn't logged in as an administrator. Non-logged-in visits to the admin page should not prompt to take administrative actions (like upgrading a DB) without first authenticating the user as someone who has permissions to do so.

---

▼ **Change History** (2)


Oldest first  Newest first  
 Show only comment text  
 Show only commits/attachments

---

 [@ocean90](#) (Core Committer) — [82 minutes ago](#) comment:1

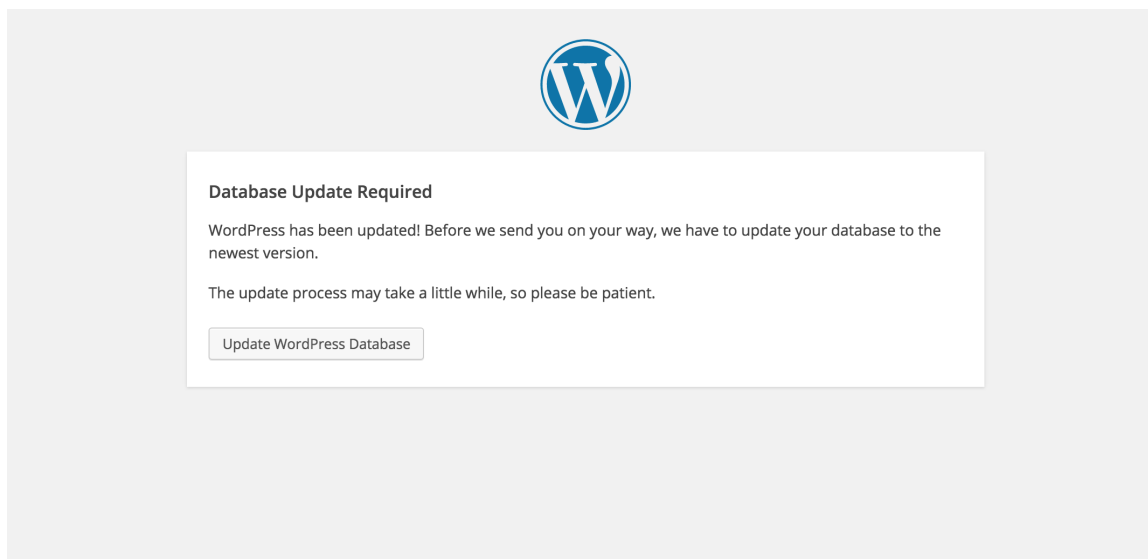
Related/Duplicate: [#3904](#)

---

 [@SergeyBiryukov](#) (Core Committer) — [75 minutes ago](#) comment:2

- **Component** changed from *General* to *Database*

**Figure 4.2:** Case Study 1: Update Screen



#### 4.1.4 Using PECCit

In another browser (and signed out), the researcher activated the Xdebug chrome extension to *debug*. The main goal was to view the execution of a visitor to */wp-admin/*, specifically the execution path and not the variable data. Thus in the Session Manager, the researcher enabled a deep Step Path Depth and turned off variable tracing and capturing. Next, the user turned on the PECCit system and visited */wp-admin/* in the separate browser (as an anonymous visitor) and verified that PECCit is running in the Session Manager. The request finished after 34 seconds and another request immediately went through the system. PECCit tracks this second request automatically and it finished after 44 seconds (see Figure 4.3). The first page, as seen through the browser, appeared to have redirected to the second. After running, the PECCit Inspector was used on the first request. Normally if a visitor goes to *wp-admin* as an anonymous user, it would ask them to log in. Instead since





the database needed to be updated, the page was redirected and asked the user to update. The next question was to see where it made the decision to redirect the user.







Since PECCit sees the entire execution, one can quickly jump to the final steps of the execution using Step Over (see Figure 4.4). Here the code path entered an *elseif* clause since apparently the *db\_version* option didn't match the *wp\_db\_version* variable (most likely setup in initialization). As a new WordPress developer, the user might note that apparently WordPress keeps track of what version it's currently using and it makes sure that the database is on that version. Since the versions didn't match, the execution decided to redirect the user to the *upgrade.php* page. Right here (or in the *elseif* clause), the code really should check if the user is logged in. Looking in the documentation, there is a function *current\_user\_can(\$capability)* which checks if the current user to the page can perform a capability. Perhaps the *update-core* capability could be used here. If the user is logged in and has the privilege to update core, then they can update the database. Inserting this conditional would probably fix this bug and make the system more secure when performing updates. Even more secure would be to check if the user is logged in on the redirected site as well. Using the PECCit Inspector on the second session (the redirected page), one can see (see Figure 4.5) that WordPress doesn't check if the user is logged in. Perhaps before including the *update.php* file and displaying the option to the user, the framework should check if the user is logged in.

#### **4.1.5 Analysis**

PECCit was useful in this case study because it allowed the developer to quickly view the execution path. The user didn't have to dig through source code and guess what would

**Figure 4.3: Case Study 1: Session Manager**

Sessions  

SID	Query	Timestamp	Status	Steps	Time	Select Time	Insert Time	Messages	Inserts	DB Incr (MB)	Settings	Error Msg	Actions
1	/research/casestudy1/wordpress/wp-admin/	10/7/15 10:04:18 PM	Done Collecting	30920	00:00:34	00:00:01	00:00:28	61843	31103	3.75000	Capture_On : false;Exclude Address : true;Property Max Children : 75;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 100;Whitelist : NoWhiteList;		  
2	/research/casestudy1/wordpress/wp-admin/...	10/7/15 10:04:52 PM	Done Collecting	34422	00:00:44	00:00:04	00:00:37	68847	34551	4.25000	Capture_On : false;Exclude Address : true;Property Max Children : 75;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 100;Whitelist : NoWhiteList;		  

be executed. It could be argued that a standard debugger could have been used to perform this walkthrough of the execution path. However if the user steps too far or if the developer accidentally clicks to perform the database update, then they would have to reinstall everything and do it over again since standard debuggers do not save debugging sessions. With PECCit, the user just had to visit the link and everything was saved to a session. Even if they accidentally clicked to update, the session is saved so they can walk through it whenever they had time or if they wanted to show someone else. Also, perhaps the user wasn't expecting the redirect and wants to analyze the redirect session. PECCit automatically records everything so the redirect was saved as well. A traditional debugger might not be configured to debug the second, unexpected request. In this example, PECCit was useful for understanding the execution path and moving forward in time very quickly without the fear of a "step over" command overstepping the mark.

**Figure 4.4:** Case Study 1: First Session

## Source Code

```
research/casestudy1/wordpress/wp-admin/admin.php ▾
036 | flush_rewrite_rules();
037 | update_option( 'db_upgraded', false );
038 |
039 | /**
040 |  * Fires on the next page load after a successful DB upgrade.
041 |  *
042 |  * @since 2.8.0
043 |  */
044 | do_action( 'after_db_upgrade' );
045 | } elseif ( get_option('db_version') != $wp_db_version && empty($_POST) ) {
046 |     if ( !is_multisite() ) {
047 |         wp_redirect( admin_url( 'upgrade.php?
_wp_http_referer=' . urlencode( wp_unslash( $_SERVER['REQUEST_URI'] ) ) ) );
048 |         exit;
049 |
050 |     /**
051 |      * Filter whether to attempt to perform the multisite DB upgrade routine.
052 |      *
053 |      * In single site, the user would be redirected to wp-admin/upgrade.php.
054 |      * In multisite, the DB upgrade routine is automatically fired, but only
055 |      * when this filter returns true.
056 |      *
057 |      * If the network is 50 sites or less, it will run every time. Otherwise,
058 |      * it will only run if the network is less than 50 sites. If the network is
059 |      * more than 50 sites, it will only run if the network is less than 50 sites.
```

Step Forward


Steps




**Figure 4.5:** Case Study 1: Second Session

## Source Code

```
research/casestudy1/wordpress/wp-admin/upgrade.php ▾
002 | /**
003 |  * Upgrade WordPress Page.
004 |  *
005 |  * @package WordPress
006 |  * @subpackage Administration
007 |  */
008 |
009 | /**
010 |  * We are upgrading WordPress.
011 |  *
012 |  * @since 1.5.1
013 |  * @var bool
014 |  */
015 | define( 'WP_INSTALLING', true );
016 |
017 | /** Load WordPress Bootstrap */
018 | require( dirname( dirname( __FILE__ ) ) . '/wp-load.php' );
019 |
020 | nocache_headers();
021 |
022 | timer_start();
023 | require_once( ABSPATH . 'wp-admin/includes/upgrade.php' );
```

Steps    

## 4.2 Case Study 2: Missing Logo on Theme

### 4.2.1 Background

WordPress works by having a main core (which handles management of content, users, pages, routing, etc), plugins (which give additional functions like e-commerce, widgets, etc), and a theme. The theme is responsible for the basic appearance of the site. Often, themes will allow users to edit their logo and site title. Many themes are free and are created and maintained by the community. In the WordPress support forum, a user asked about a free theme which was having an issue.<sup>6</sup>

### 4.2.2 Problem

In the forum post (see Figure 4.6), the user was finding that they could upload a logo to be used by a theme called Trident-Lite<sup>7</sup>, but the logo wasn't appearing when they looked at the site. Oddly, they saw the logo when they previewed the site from the administrator panel, but not when they viewed the actual site. The question/issue had initially been asked 6 months prior to this case study but no one responded to the inquiry. An additional user responded a couple days prior to the study asking if the user had found a way to fix the bug indicating that the bug could still be present.

---

<sup>6</sup><https://wordpress.org/support/topic/logo-missing-2>

<sup>7</sup><https://wordpress.org/themes/trident-lite/>

**Figure 4.6: Case Study 2: Forum Post**

Forums    [Log in](#) ([forgot?](#)) [Register](#)


---

[WordPress](#) • [Support](#) • [Themes and Templates](#)

### Trident Lite


Logo missing (2 posts)

---

 **mecrowe1970**  
Member  
Posted 6 months ago #

Using Trident theme, locally. When I use the customizer to add a logo, it shows up in the preview to the right fine. However, when I go into the editor and work and then view the page, nothing...Any suggestions? Ive tried stopping any plugins, have removed the image and uploaded it again, checked the format (jpg) and size (775KB) with no success

---

 **pozzana**  
Member  
Posted 4 days ago #

I have the same problem. Have you soved yhis?

---


**Reply**  
You must [log in](#) to post.

---

**About this Theme**

[Trident Lite](#)  
[Support Threads](#)  
[Reviews](#)

**About this Topic**

 [RSS feed for this topic](#)

Started 6 months ago by mecrowe1970

[Latest reply from pozzana](#)

This topic is not resolved

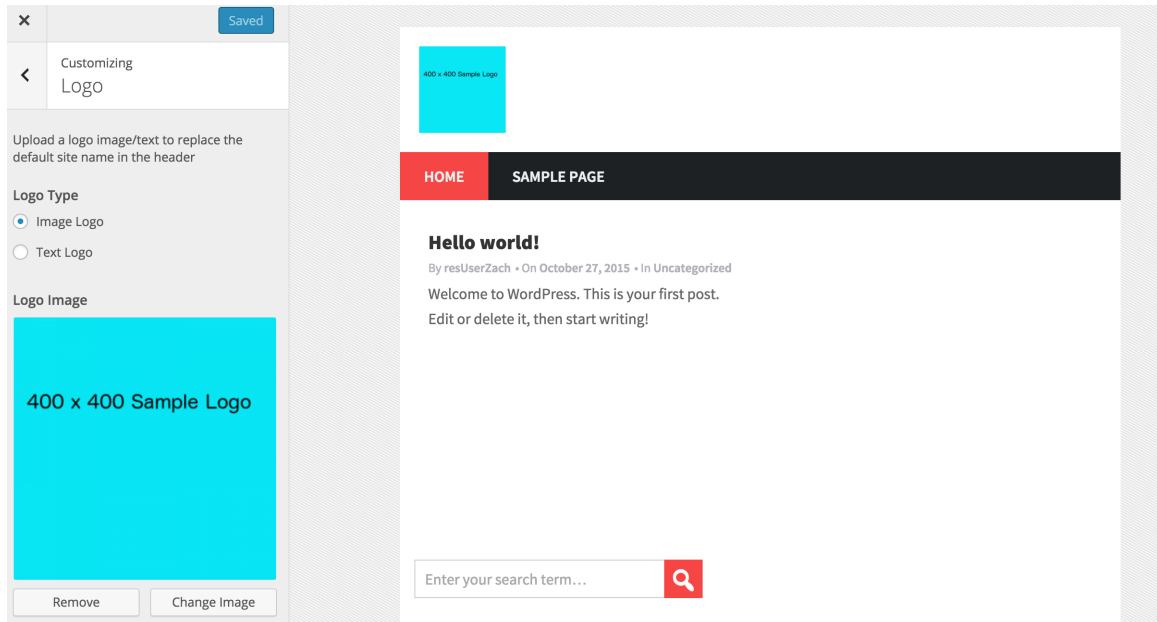
WordPress version: 4.1.1

---

**Tags**

---

**Figure 4.7:** *Case Study 2: Preview With Image*



### 4.2.3 Setup

A fresh install of a WordPress site was created on the research server. The Trident-Lite theme was installed, a logo image was added, and the modifications were saved. This successfully recreated the bug. One was able to see the logo image in the preview (see Figure 4.7) but when the main site was visited, the image was not there (see Figure 4.8).

### 4.2.4 Using PECCit

First, the code responsible for printing the logo needed to be found. As the logo was part of the header of the page, the researcher looked in a file called *header.php* and, by browsing through the source code on the command line, found the code which would print the logo

**Figure 4.8:** Case Study 2: Home Page Without Image

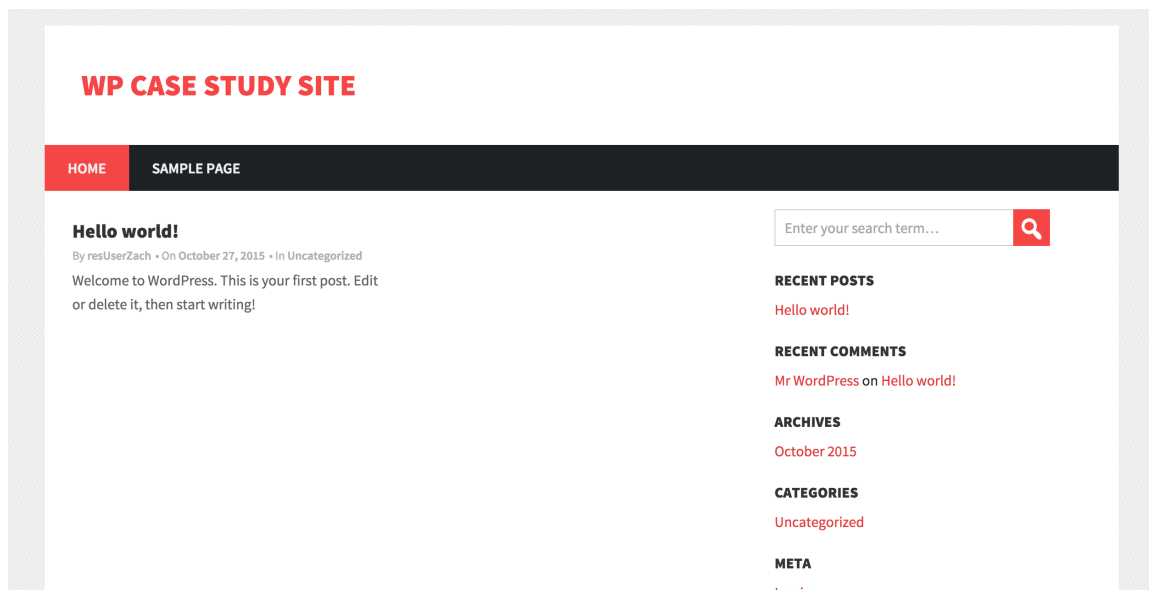


image (see Figure 4.9). Tracing is expensive so finding this file allowed the trace to be fairly specific by listing the *header.php* file on the whitelist. It appeared that the logo would only print based on certain theme settings that were retrieved using the *get\_theme\_mod()* function. Looking in the WordPress documentation, this function belongs to a core WordPress file called *theme.php*. This file contains functions which are useful to theme developers (including the *get\_theme\_mod()* and *set\_theme\_mod()* functions which can be used to get/set theme settings in the database). This WordPress core file was whitelisted as well. For the rest of the settings, capture was turned off, a deep step max was set, and a reasonable max children and property max depth were used. Then with debugging enabled and the ADS running, the researcher visited the home page.

The trace required a little over 3 minutes and completed successfully. From looking at the source code earlier, line 72 of the *header.php* file appeared to be the beginning of the



**Figure 4.9:** Case Study 2: Header File

```
69 <div class="header-logo">
70 <a class="skip-link screen-reader-text" href="#content"><?php _e( 'Skip to content', 'qlueblog' ); ?></a>
71 <a class="brand" href='<?php echo esc_url( home_url( '/' ) ); ?>' title='<?php bloginfo( 'name' ); ?>' rel="home"> _____
72 <?php if ( get_theme_mod( 'trident_logo' ) || get_theme_mod( 'trident_logo_text' ) ) : ?>
73 <?php if ( 'image' == get_theme_mod( 'trident_logo_type' ) ) : ?>
74 <img src='<?php echo esc_url( get_theme_mod( 'trident_logo' ) ); ?>' alt='<?php echo esc_attr( get_bloginfo( 'name' ),
75 <?php else : ?>
76 <span class="logo-text"><?php if( get_theme_mod( 'trident_logo_text' ) ) { echo get_theme_mod( 'trident_logo_text' ); } e
77 <?php endif; ?>
78 <?php else : ?>
79 <span><?php bloginfo( 'name' ); ?></span> _____
80 <?php endif; ?> _____
81 </a>
82 </div>
```

*if* statements for printing the logo (see Figure 4.9). Thus, this was a good place to start debugging. The Step Finder (see Figure 4.10) was used to directly jump to that line in the execution. At this point, Step Into was used to discover that *get\_theme\_mod( "trident\_logo" )* correctly returned a file name, but *get\_theme\_mod( "trident\_logo\_type" )* didn't return "image" as the *trident\_logo\_type* theme modification wasn't specified (see Figure 4.11). Thus, the line of execution goes to the *else* clause and the *logo-text* is printed.

Using this information, it appears that the theme settings weren't correctly saved when the logo was uploaded. The user quickly gained valuable insight into the bug. On a hunch, the researcher went back to the administrator panel where the logo image could be selected. To test, the text logo option was toggled and saved. Then the image logo option was toggled and saved. It was presumed that this specifically told the theme to use the logo image. When visiting the main page, it appeared that this fixed the bug as the image was correctly being displayed (see Figure 4.12).

To see what happened, the same trace was performed now that it was working. It required about the same amount of time. Doing the same actions as before in the PECCit Inspector, it was confirmed that this time *get\_theme\_mod( "trident\_logo\_type" )* returned "image" (see Figure 4.13). Thus, the logo was printed to the screen and the side effect of

Figure 4.10: Case Study 2: Step Finder

Source Code

```
research/casestudies/wp/wpTheme/wp-content/themes/trident-lite/header.php
870 |         <a class="skip-link screen-reader-text" href="#content">?
871 |     <php _e( 'Skip to content', 'qLublog' ); ?></a>
872 |     <a class="home" href="#"><php echo esc_url( home_url( '/' ) ); ?> title=<?
873 |         <?php if ( get_theme_mod( 'trident_logo' ) || get_theme_mod( 'trident_logo_text' ) ) : ?>
874 |             <img src=<?php echo esc_url( get_theme_mod( 'trident_logo' ) ); ?> alt=<?
875 |     <?php else : ?>
876 |         <span class="logo-text"><?
877 |     <?php endif; ?>
878 |     <?php else : ?>
879 |         <span><?php bloginfo( 'name' ); ?</span>
880 |     <?php endif; ?>
881 | </a>
882 | </div>
883 | <?php if ( !dynamic_sidebar( 'header-widget' ) ) : ?>
884 | <?php endif; ?>
885 | <div class="clearfix"></div>
886 | </div>
887 | <div class="row">
888 |     <nav id="site-navigation" class="main-navigation" role="navigation">
889 |         <div class="menu-toggle">
890 |             <button class="nav-bar-toggle collapsed" type="button" data-toggle="collapse" data-target=".bs-
891 |                 <span class="sr-only">
892 |                     <?php _e( 'Toggle navigation', 'trident' ); ?>
893 |                 </span>
894 |                 <span class="icon-bar"></span>
895 |                 <span class="icon-bar"></span>
```

Steps ▲ ◀ ▶ ▼

Variables Query Info Search Step Finder Captures

Find All Steps at Line Number

Line Number (Optional)

72

File

research/casestudies/wp/wpTheme/wp-content/themes/trident-lite/header.php

Find Steps

Results:

Step: 474694

Step: 474695

the bug was fixed. Note here that the bug itself (not correctly saving the logo settings when installed) was not fixed.

## 4.2.5 Analysis

PECCit was valuable in this case study because it allowed one to quickly jump to a line of code in the execution, see what happened there, and what the variable values were. This is similar to a breakpoint in a standard debugger. What's powerful here though is that the debugging sessions are saved and show what the variables were at that time (like the theme modification settings). This debugging session could be shared with the theme developer to give them insight into how to fix the bug. As a PECCit user, the researcher was able to quickly find the bug, fix it temporarily, and gain insight into the bug's location to tell the theme developer without really needing to understand how WordPress works or how

Figure 4.11: Case Study 2: Modification Settings Before Fix

The screenshot shows a code editor with PHP code for theme modifications. The code includes comments and function definitions for setting theme modification values. A variables panel on the right lists the following variables:

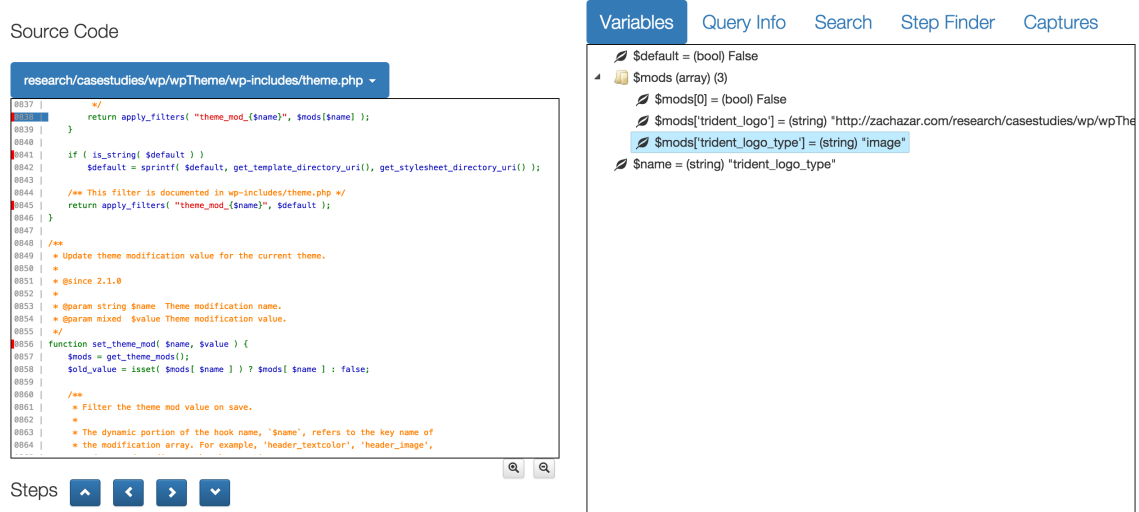
- \$default = (bool) False
- \$mods (array) (2)
  - \$mods[0] = (bool) False
  - \$mods['trident\_logo'] = (string) "http://zachazar.com/research/casestudies/wp/wpThem"
- \$name = (string) "trident\_logo\_type"

Notice above that the \$mods array does not contain a 'trident\_logo\_type' key

Figure 4.12: Case Study 2: Home Page With Image

The screenshot shows a WordPress home page. The navigation bar includes "HOME" and "SAMPLE PAGE". The main content area displays "Hello world!" with a byline "By resUserZach • On October 27, 2015 • In Uncategorized" and a welcome message: "Welcome to WordPress. This is your first post. Edit or delete it, then start writing!". The sidebar on the right contains a search bar and sections for "RECENT POSTS", "RECENT COMMENTS", "ARCHIVES", "CATEGORIES", and "META".

**Figure 4.13:** Case Study 2: Modification Settings After Fix



the theme was designed. Also since PECCit is an omniscient debugger, one could step backward and forward in time without worrying about overstepping. With the ability to whitelist the trace, a partial trace was performed of only the files of interest which saved time and storage.

## 4.3 Case Study 3: Duplicate Stores In Plugin

### 4.3.1 Background

As mentioned in Case Study 2: Missing Logo on Theme, WordPress allows the use of plugins to extend the core functionality of the framework. One plugin is called WP Store Locator<sup>8</sup>. It allows the site administrator the ability to add *stores* to the site and the users

<sup>8</sup><https://wordpress.org/plugins/wp-store-locator/>

can search for stores near their location. The plugin displays the nearby stores on a Google Map embedded on the page.

### 4.3.2 Problem

A user wrote on the support forum (see Figure 4.14) that they were seeing duplicate entries for stores on their page.<sup>9</sup> They wrote that the same store will sometimes show up four different times in the search list and, even though the duplicate stores are at the same location, they show different distances. Before starting the case study when navigating to the user's site (an all-natural fertilizer site), it was confirmed that entries were showing up four times (though unfortunately, a screenshot wasn't taken of the user's site before they fixed it).

### 4.3.3 Setup

The WP Store Locator plugin was installed on a test WordPress site. A test *store* was added to the plugin and the location was specified. When navigating to the browser, one could see the test store in the results list as well as on the map (see Figure 4.15). The store only showed up once however. The settings were adjusted trying to recreate the duplication problem that the user was experiencing but unfortunately the duplication problem could not be reproduced. Thus, the purpose of this case study is to demonstrate what a developer might do if they were seeing a duplication problem.<sup>10</sup> Using PECCit on the client's system

---

<sup>9</sup><https://wordpress.org/support/topic/multiple-copies-of-stores-returned-in-search>

<sup>10</sup>It's often difficult to exactly recreate a problem that someone is experiencing and describing on a support post without access to the database and the code (another plugin or other custom code could be interfering).

**Figure 4.14:** *Case Study 3: Forum Post*

## WP Store Locator

Multiple copies of stores returned in search (2 posts)



**dtesysop**  
Member  
Posted 1 week ago #

Hi,

I am receiving multiple copies of stores being returned via a search. There are usually four copies, but each of them is listed as having a different distance from the search criteria, anywhere from a little less than a quarter mile to more than a mile, though they all show the same address. It's not for every store returned, just a handful of them. My URL is:

<http://downtoearthfertilizer.com/where-to-buy/>

<https://wordpress.org/plugins/wp-store-locator/>



**Tijmen Smit**  
Member  
Plugin Author

Posted 1 week ago #

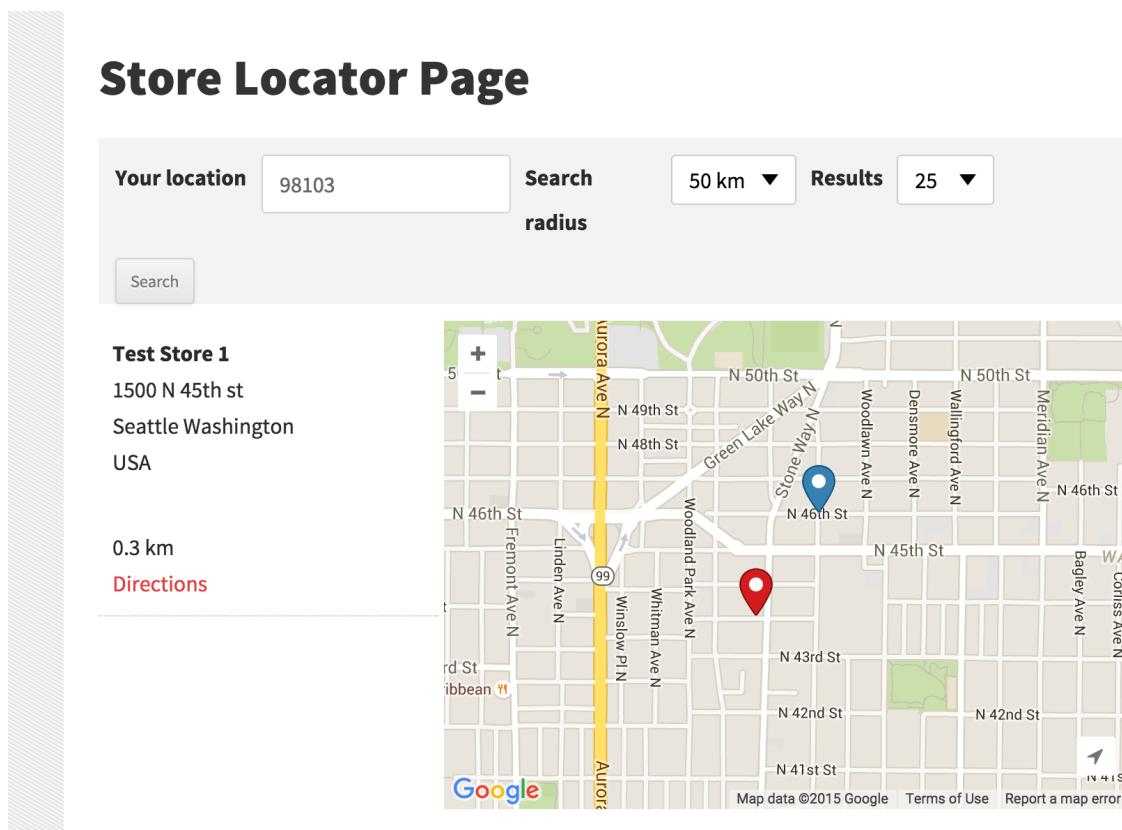
This is a different problem than what others have. You aren't using WPML right? The location ids are all the same, and so are the latlng values.

I checked the JSON response, and if you search for 'Eugene' it shows the same data 4 times. So it looks like the sql query somehow decided to return the same data 4 times, no idea why that would happen, and why it doesn't happen for all locations.

Have you tried deleting the one who shows 4 times, and create a new one with the same data?

How are your tech skills? If I would ask you to run a sql query in phpmyadmin would that work?

Figure 4.15: Case Study 3: Test Plugin with Test Store



could alleviate this problem because the traces can be saved and shared with others (even if the technical team nor developer can't access the database nor recreate the issue).

#### 4.3.4 Using PECCit

First, a deep step trace (without capturing nor variable tracing) was performed. The goal was to see what lines of code were executed using the Execution Path Highlighting feature of PECCit. When traced, surprisingly two requests came into the system (see Sessions 1 and 2 in Figure 4.16). Since PECCit automatically records everything coming in, both

**Figure 4.16: Case Study 3: Sessions**

Sessions  

SID	Query	Timestamp	Status	Steps	Time	Select Time	Insert Time	Messages	Inserts	DB Incr (MB)	Settings
1	/research/casestudies/wp/wpTheme/index.p...	11/8/15 5:05:54 PM	Done Collecting	85153	00:00:39	00:00:03	00:00:27	170309	85364	9.00000	Capture_On : false;Exclude Address : true;Property Max Children : 75;Property Max Depth : 5;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : nowhitelist;
2	/research/casestudies/wp/wpTheme/wp-admi...  /research/casestudies/wp/wp-admin/admin-ajax.php?action=store_search&lat=47.60621&lng=-122.33207099999998&max_results=25&radius=50&autoload=1	11/8/15 5:06:34 PM	Done Collecting	43742	00:00:23	00:00:03	00:00:15	87487	43905	3.03125	Capture_On : false;Exclude Address : true;Property Max Children : 75;Property Max Depth : 5;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : nowhitelist;
3	/research/casestudies/wp/wpTheme/wp-admi...	11/8/15 5:15:09 PM	Done Collecting	48301	00:00:55	00:00:13	00:00:33	108670	122419	11.15625	Capture_On : true;Exclude Address : true;Property Max Children : 75;Property Max Depth : 5;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : wp-store-locator;

requests were recorded (which is a very nice feature since it was not known that the second request was going to come. A traditional debugger most likely would not break on this request). The first request appeared to load the appearance of the page while the second request, an AJAX request from the browser, searched for store locations and returned them. Since the AJAX request dealt with the the list of stores and the support post referred to duplicates in the list of stores, then the AJAX request was most likely where the bug would be.

The PECCit Inspector was used to analyze the AJAX request. Instead of following the lines of execution, the researcher wanted to quickly see where in the code the data (stores) were loaded. The user clicked on the source files to quickly scan the Execution Path Highlighting. After trying a couple files, the *frontend/class-frontend.php* file was examined with the presumption that this could perhaps be a frontend issue. Quickly scrolling, one



could see that most of the coverage was in a *store\_search()* function. A comment above it read “Handle the Ajax search on the frontend” indicating this might be near the bug (see Figure 4.17). At the end of the function, there was a line of code that was sending (as JSON) a variable called *\$store\_data*. It would be handy to see the value for *\$store\_data*, but the file was not whitelisted for this session. Since this was an AJAX request and the number of steps were low for the session, it was decided to retry the AJAX request but with the whitelist changed to trace the entire plugin. With this setting, all execution in the plugin would be traced. On the test site, a “search” was performed for nearby stores to fire the AJAX request again. After about one minute, the session was done (see Session 3 in Figure 4.16).

With this new session, there was no need to waste any time to get to that same line of code. To quickly jump there, the Step Finder was used to look for that line of code in the file (see Figure 4.18). Next, the Variables tab for the *\$store\_data* variable was examined. Here, one could see that it was an array containing the one store that was expected (see Figure 4.19). If one were the site administrator having the duplication problem, this would be a great place to check if there were four items for each single item.

Let’s say that there were four items in *\$store\_data*. Next, one would need to find out where that *\$store\_data* array is loaded. The developer could search through source code and, if they were using a traditional debugger, re-fire the AJAX request and break on that initialization line hoping they haven’t over stepped, but they don’t need to do any of that with PECCit. All one needs to do is right-click the variable and *Inspect* (see Figure 4.20). In the inspection results (see Figure 4.21), the first step is where the variable initially came into scope and is uninitialized. The second step is more interesting because this is where

Figure 4.17: Case Study 3: Store Search

```
locator/frontend/class-frontend.php ▾
0066 |      /**
0067 |       * Handle the Ajax search on the frontend.
0068 |       *
0069 |       * @since 1.0
0070 |       * @return json A list of store locations that are located within the selected search radius
0071 |       */
0072 |     public function store_search() {
0073 |
0074 |         global $wpsl, $wpsl_settings;
0075 |
0076 |         /* Check if auto loading the locations on page load is enabled.
0077 |          *
0078 |          * If so then we save the store data in a transient to prevent a long loading time
0079 |          * in case a large amount of locations need to be displayed.
0080 |          *
0081 |          * The SQL query that selects nearby locations doesn't take that long,
0082 |          * but collecting all the store meta data in get_store_meta_data() for hundreds,
0083 |          * or thousands of stores can make it really slow.
0084 |          */
0085 |
0086 |         if ( $wpsl_settings['autoload'] && isset( $_GET['autoload'] ) && $_GET['autoload'] && !$wpsl_settings['debu
0087 |
0088 |             /* If a multilingual plugin ( WPML or qTranslate X ) is active then we have
0089 |              * to make sure each language has his own unique transient. We do this by
0090 |              * including the lang code in the transient name.
0091 |              *
0092 |              * Otherwise if the language is for example set to German on page load,
0093 |              * and the user switches to Spanish, then he would get the incorrect
0094 |              * permalink structure ( /de/. instead of /es/. ) and translated
0095 |              * store details.
0096 |              */
0097 |             $lang_code = $wpsl->i18n->check_multilingual_code();
0098 |
0099 |             if ( false === ( $store_data = get_transient( 'wpsl_autoload_' . $wpsl_settings['autoload_limit'] . $la
0100 |                 $store_data = $this->find_nearby_locations();
```

Figure 4.18: Case Study 3: Step Finder

The screenshot displays the Step Finder tool interface. On the left, the 'Source Code' panel shows PHP code from the file `research/casestudies/wp/wpTheme/wp-content/plugins/wp-store-locator/frontend/class-frontend.php`. The code includes a function `find_nearby_locations()` that checks for a transient cache, queries the database for nearby stores, and returns the results as an array. On the right, the 'Step Finder' panel is active, showing 'Find All Steps at Line Number' with a search box containing '109'. Below this, the 'File' dropdown shows the current file path. A 'Find Steps' button is visible, and the 'Results' section shows 'Step: 177077'.

it became an array. Clicking on the second step, PECCit shows earlier in time when the `$store_data` variable was initialized to an array in a `find_nearby_locations()` function. Here, one could step forward to see how the plugin developer queried the database to find nearby locations. On line 208, one can see that the database is queried (through WordPress's database abstraction) for the nearby stores (see Figure 4.22). The developer can look at the `$stores` variable to see what is returned from the query. In this case, it is the single store. This would be an important place for the user with the duplication problem to see if there are duplicate entries in `$stores`. If there are, the error is most likely caused by a corrupt database, an incorrect `SELECT` statement, or the results are corrupted during the abstraction process (maybe from another plugin) since the results are directly returned from querying through the database abstraction. If there are not duplicate locations, there could be an error caused by line 211 where filters are altering the data.

Figure 4.19: Case Study 3: \$store\_data Array

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/wp-store-locator/frontend/class-frontend.php
0097 |
0098 |
0099 |     if ( false === ( $store_data = get_transient( 'wpsl_autoload_' . $wpsl_settings['autoload_limit'] . $lang_code
$store_data = $this->find_nearby_locations();
0100 |
0101 |         if ( $store_data ) {
0102 |             set_transient( 'wpsl_autoload_' . $wpsl_settings['autoload_limit'] . $lang_code, $store_data, 0 );
0103 |         }
0104 |     }
0105 |     } else {
0106 |         $store_data = $this->find_nearby_locations();
0107 |     }
0108 |
0109 |     wp_send_json( $store_data );
0110 |
0111 |     exit();
0112 |
0113 |
0114 |
0115 |     /**
0116 |     * Find store locations that are located within the selected search radius.
0117 |     * This happens by calculating the distance between the
0118 |     * latlng of the searched location, and the latlng from
0119 |     * the stores in the db.
0120 |     * @since 2.0
0121 |     * @return void|array $store_data The list of stores that fall within the selected range.
0122 |     */
0123 |     public function find_nearby_locations() {
0124 |
0125 |         global $wpdb, $wpsl, $wpsl_settings;
0126 |
0127 |

```

Steps

Variables Query Info Search Step Finder Captures

```

$lang_code (uninitialized)
$store_data (array) (1)
  $store_data[0] (array) (17)
    $store_data[0]['address'] = (string) "1500 N 45th st"
    $store_data[0]['address2'] = (string) ""
    $store_data[0]['city'] = (string) "Seattle"
    $store_data[0]['country'] = (string) "USA"
    $store_data[0]['distance'] = (float) 0.3
    $store_data[0]['email'] = (string) ""
    $store_data[0]['fax'] = (string) ""
    $store_data[0]['hours'] = (string) "<table class='wpsl-opening-hours'><tr><td>Mo
    $store_data[0]['id'] = (string) "10"
    $store_data[0]['lat'] = (string) "47.662147"
    $store_data[0]['lng'] = (string) "-122.340619"
    $store_data[0]['phone'] = (string) ""
    $store_data[0]['state'] = (string) "Washington"
    $store_data[0]['store'] = (string) "Test Store 1"
    $store_data[0]['thumb'] = (string) ""
    $store_data[0]['url'] = (string) ""
    $store_data[0]['zip'] = (string) ""

```

Figure 4.20: Case Study 3: Variable Inspector Tool

Variables
Query Info
Search
Step Finder

```

$lang_code (uninitialized)
$store_data (array) (1)
  $store_data[0] (array) (17)
    $store_data[0]['address'] = (string) "1500 N 45th st"
    $store_data[0]['address2'] = (string) ""
    $store_data[0]['city'] = (string) "Seattle"
    $store_data[0]['country'] = (string) "USA"

```

Figure 4.21: Case Study 3: Results of Variable Inspector

Variable: \$store_data						
Location First Set	Value	Type	Classname	Facet	Number of Children/Properties	
172400		uninitialized				
172405		array			0	
177076		array			1	

Figure 4.22: Case Study 3: \$stores Array

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/wp-store-locator/frontend/class-frontend.php
8190 |         AS distance
8200 |         FROM $wpdb->posts AS posts
8201 |         INNER JOIN $wpdb->
8202 |         AS post_lat ON post_lat.post_id = posts.ID AND post_lat.meta_key = 'wpl_lat'
8203 |         INNER JOIN $wpdb->
8204 |         AS post_lng ON post_lng.post_id = posts.ID AND post_lng.meta_key = 'wpl_lng'
8205 |         WHERE posts.post_type = 'wpl_stores'
8206 |         AND posts.post_status = 'publish' $group_by $sql_sort"
8207 |
8208 |         $stores = $wpdb->get_results( $wpdb->prepare( $sql, $placeholder_values ) );
8209 |
8210 |         if ( $stores ) {
8211 |             $store_data = apply_filters( 'wpl_store_data', $this->get_store_meta_data( $stores ) );
8212 |
8213 |         }
8214 |         return $store_data;
8215 |     }
8216 |
8217 |     /**
8218 |      * Get the post meta data for the selected stores.
            
```

Variables	Query Info	Search	Step Finder	Captures
<ul style="list-style-type: none"> <li>\$cat_filter = (string) ""</li> <li>\$group_by = (string) ""</li> <li>\$limit (uninitialized)</li> <li>\$placeholder_values (array) (6)</li> <li>\$radius = (int) 6371</li> <li>\$sql = (string) "SELECT post_lat.meta_value AS lat, post_lng.meta_value AS lng, posts.ID</li> <li>\$sql_sort = (string) "HAVING distance &lt; %d ORDER BY distance LIMIT 0, %d"</li> <li>\$stores (array) (1)                             <ul style="list-style-type: none"> <li>\$stores[0] (stdClass) (4)                                     <ul style="list-style-type: none"> <li>\$stores[0]-&gt;distance = (string) "0.278585799997171"</li> <li>\$stores[0]-&gt;ID = (string) "10"</li> <li>\$stores[0]-&gt;lat = (string) "47.662147"</li> <li>\$stores[0]-&gt;lng = (string) "-122.340619"</li> </ul> </li> </ul> </li> <li>\$store_data (array) (0)</li> <li>\$this (WPSL Frontend) (3)</li> </ul>				

### 4.3.5 Analysis

This case study highlights a number of powerful tools that the PECCit system offers. First, the automatic tracing was able to capture the second, unexpected AJAX request. Next, the Execution Path Highlighting feature allowed for quickly seeing what code was executed in the plugin without being familiar with the plugin or WordPress AJAX handling. Next, it was easy to rerun the AJAX request but with different settings (specifically, variable tracing). The sessions required very little time (under 1 minute each) since the whitelist was specific. Next, Step Finder was used to quickly jump to the line of code that seemed interesting. One could argue that this process was similar to a standard debugging session where one starts debugging, sets a breakpoint into the future, then “resumes” until the breakpoint is hit. However in this scenario, the user wanted to go to that line of code with the intention of going backwards after that (something standard debuggers don’t offer). With the Variable Inspector and the tools that omniscient debugging give, one was able to go back in time to see when and where the *\$store\_data* variable was initialized and populated. Not only that, but the researcher was able to see the values of other variables like *\$stores* which previously weren’t of interest. Lastly as with the other case studies, these sessions can be saved and shared with other developers. They can even be viewed and walked through at the same time by other developers since the application is web based. Thus, recreation on the developer’s side isn’t necessary if the client can capture the faulty behavior in a single trace on their side.

## 4.4 Case Study 4: Incorrect View Count Plugin

### 4.4.1 Background

WordPress can get fairly complicated under the hood with all of the different plugin/theme options for interacting with page creation, interrupting the system, and altering the framework's work process. Hooks are a standard way for plugins and templates to interact with pages being built. Hooking<sup>11</sup> is a way for the developer to attach to / remove code from a particular event (action hook) or alter data fired with a particular event (filter hook). For example when WordPress's core is building the footer, it fires a *wp\_footer* event that a plugin or theme can hook to and change the footer's appearance.

### 4.4.2 Problem

A number of users in a support forum<sup>12</sup> were having problems with a plugin that they were using called Page View Count<sup>13</sup> as shown in Figure 4.23. This plugin allows the site administrator to attach statistics to a page showing the visitors how many times a page has been viewed (that day and all time) as well as a number of other features/settings. The forum users explained that the count was showing incorrect values and displaying oddly on the home page but it was working fine on other pages. Thus for whatever reason, it was malfunctioning on the home page.

---

<sup>11</sup><http://codex.wordpress.org/Glossary#Hook>

<sup>12</sup><https://wordpress.org/support/topic/counts-disappeared>

<sup>13</sup><https://wordpress.org/plugins/page-views-count/>


Figure 4.23: Case Study 4: Forum Post

Forums     (forgot?)

WordPress · Support · Plugins and Hacks

## Page View Count

Counts disappeared (4 posts)




**sanderfeinberg**  
Member  
Posted 2 weeks ago #

Counts reset/disappeared on home page while still appearing elsewhere. No changes were made to home page.

<http://rohonasurvivors.org>

<https://wordpress.org/plugins/page-views-count/>


---



**li9ssb**  
Member  
Posted 1 week ago #

Disable all your plugins.  
Enable only Page View.  
Now go enabling one by one to see which is the one that prevents the action.


---



**Milly Moll**  
Member  
Posted 6 days ago #

I am having the same problem. Every time you go into wordpress or update a page the count vanishes. This has happened ever since I updated the plugin to version 1.3.0 Before that everything was fine and I have not changed or added any new plugins since then. You can manually add a count but that would require you to keep a daily tally of your previous page counts just in case it wipes out your totals again. Tedious to say the least and renders the plugin useless.

---




**darkan9el**  
Member  
Posted 1 day ago #

Same here, this has nothing to do with other plugins, I installed on a clean new install of wordpress with this being the only plugin and it still does it. Everything after the comma gets removed. you can remove the comma, update and it adds a comma. update again with a comma already there and it removed the digits to the right of the comma and also the comma vanishes. Any number up to 999 will be ok but put a comma in e.g 9,99 or 99,9 or ,999 and it goes wrong the ,999 just defaults to 0

disabled all plugins on another website with bulk actions, enabled page view only and still same issue


Same as you milly updated to 1.3.0 and it all got messed up, must be a fault in the coding somewhere, this is not a plugin compatibility issue, I would suggest you compare your coding from 1.2.1 as that version worked ok for me

### About this Plugin



Page View Count  
Frequently Asked Questions  
Support Threads  
Reviews

### About this Topic

 RSS feed for this topic

Started 2 weeks ago by sanderfeinberg  
Latest reply from darkan9el

This topic is not resolved

WordPress version: 4.3.1

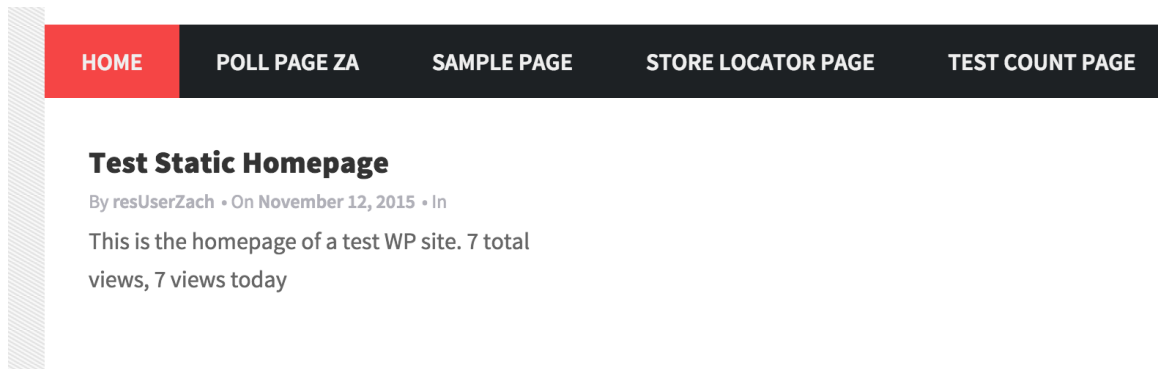
### Tags

No tags yet.

117



**Figure 4.24:** *Case Study 4: Home Page with Incorrect Formatting*



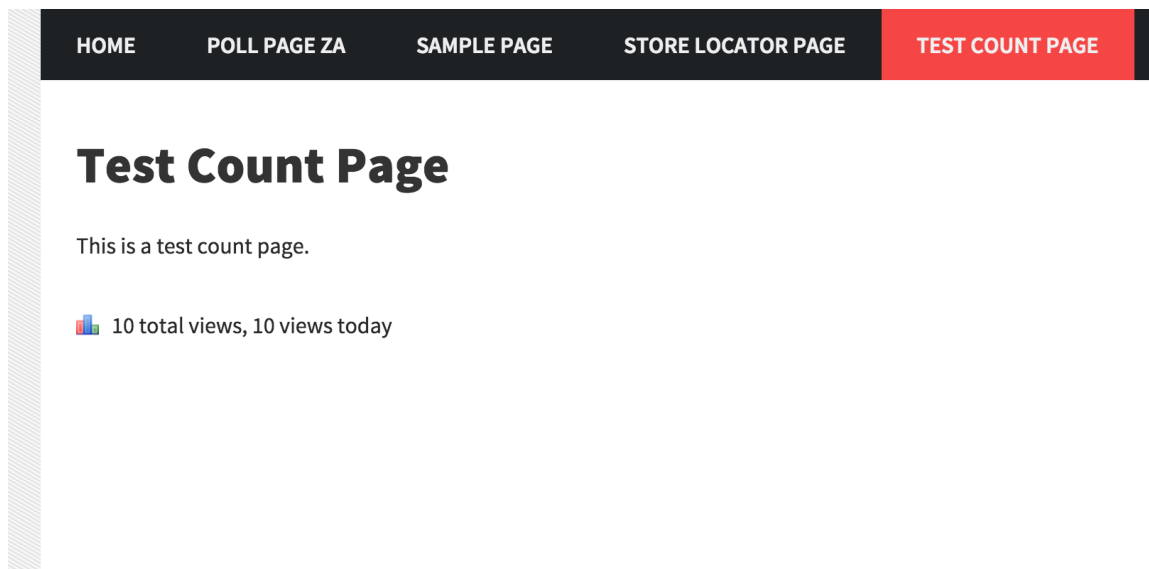
### 4.4.3 Setup

Using the test WordPress site used for the other case studies, the Page View Count plugin was downloaded and installed. The settings were altered such that a count would be displayed on every page (including the home page). It was verified that the count was incorrectly displayed on the home page (see Figure 4.24) and was not counting visits when users would visit the page. A test page (non-home page) was created and confirmed that the count was correctly displayed on this page and it was increasing correctly when users would visit the page via other browsers. (see Figure 4.25).

### 4.4.4 Using PECCit



To start, a variable trace of the home page was performed with a large step max and a whitelist of the entire plugin. The max children and property depth were moderate and capturing was turned off. The session required around 6 minutes to trace (see Session 1 in Figure 4.26). The page created during the trace had the page count issues and it showed

**Figure 4.25:** Case Study 4: Test Page with Correct Formatting



the page count as “12”. Since it was unknown in the code where there was a problem or even when the plugin was used during the execution, the Search tool was used. The variable value of “12” was searched (since that was the page count) as shown in Figure 4.27. The first variable was inspected (see Figure 4.28) which found when the variable was created (see Figure 4.29). After clicking the link, one could see that the variable was used in a function called *pvc\_get\_stats()* (see Figure 4.30). The level of execution was 13 levels down, so it would have been very difficult to find this step so quickly without the Search tool. The function appeared to be responsible for creating the HTML that would show the page count for this page. The *\$output\_html* variable (which was going to be returned) looked fine at this line of code but the last line of code appeared to apply filters to the HTML.

**Figure 4.26: Case Study 4: Sessions**

Sessions  

SID	Query	Timestamp	Status	Steps	Time	Select Time	Insert Time	Messages	Inserts	DB Incr (MB)	Settings	Error Msg
1	/research/casestudies/wp/wpTheme/	11/12/15 4:24:32 PM	Done Collecting	92274	00:06:52	00:02:21	00:03:40	328891	747514	45.17188	Capture_On : false;Exclude Address : true;Property Max Children : 50;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : page-views-count;	
2	/research/casestudies/wp/wpTheme/index.p...	11/12/15 4:35:59 PM	Done Collecting	91043	00:06:36	00:02:11	00:03:53	326429	746173	43.93750	Capture_On : false;Exclude Address : true;Property Max Children : 50;Property Max Depth : 4;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : page-views-count;	

The next question was, “do any of those filters change the HTML causing the bug?” At this point, one could step forward and manually look at the *\$output\_html* variable in hopes of spotting a difference (if one exists) but instead the Variable Differences tool was used (as described in Section 3.4.3). Thus, one could simply highlight the next Step and confirm that there were no differences made to the variable from the filtering (see Figure 4.31).

Stepping out (to get a better grasp of where the code execution had gone), the function had been called by a function called *pvc\_stats\_counter()* (see Figure 4.32) which was called by a *pvc\_stats\_show()* (see Figure 4.33). The resulting HTML being created seemed correct. To compare this HTML, a trace of the test page where the count was being displayed correctly was performed (see Session 2 in Figure 4.26). Using the Step Finder (see Figure 4.34), the line where the HTML was created was instantly jumped to and it looked identically formatted to that created by the home page (see Figure 4.35). Thus, something was altering the HTML – perhaps the theme. Stepping out in the front page session, one could see that the theme was using a *front-page.php* file. Stepping out in the page session, it appeared to be using a *content-page.php* theme file to display the page. Looking at these

**Figure 4.27:** Case Study 4: Search Tool Results for "12"

Variables   Query Info   **Search**   Step Finder   Captures

Settings

Search Variable Short Name

Search Variable Full Name

Search Variable Values

Use Wildcards

Display

Display As Tree

12 Search

- `$results->today = (string) "12"`
- `$results->total = (string) "12"`
- `$wpdb->last_result[0]->today = (string) "12"`
- `$wpdb->last_result[0]->total = (string) "12"`

**Figure 4.28:** Case Study 4: Inspect Tool in Search Results

Variables Query Info **Search** Step Finder Captures

Settings

Search Variable Short Name

Search Variable Full Name

Search Variable Values

Use Wildcards

Display

Display As Tree

12 Search

- \$results->today = (string) "12"
- Inspect Variable "12"
- \$wpdb->last\_result[0]->total = (string) "12"

**Figure 4.29:** Case Study 4: Results of Inspecting \$results->today

Variables **Query Info** Search Step Finder Captures

Variable: \$results->today

Location	First Set	Value	Type	Classname	Facet	Number of Children/Properties
73458		12	string		public	

Figure 4.30: Case Study 4: Results from pvc\_get\_stats() of Home Page

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/pvc_class.php
119 |
120 |     public static function pvc_get_stats($post_id) {
121 |         global $wpdb;
122 |
123 |         $output_html = '';
124 |         // get all the post view info to display
125 |         $results = A3_PVC::pvc_fetch_post_counts( $post_id );
126 |         // get the stats and
127 |         if ( $results ) {
128 |             $output_html .= number_format( $results->total ) . '&nbsp;'; __('total views', 'page-views-
count') . ', ' . number_format( $results->today ) . '&nbsp;'; __('views today', 'page-views-count');
129 |         } else {
130 |             $total = A3_PVC::pvc_fetch_post_total( $post_id );
131 |             if ( $total > 0 ) {
132 |                 $output_html = number_format( $total ) . '&nbsp;'; __('total views', 'page-views-
count') . ', ' . __('no views today', 'page-views-count');
133 |             } else {
134 |                 $output_html = __('No views yet', 'page-views-count');
135 |             }
136 |         }
137 |         $output_html = apply_filters( 'pvc_filter_get_stats', $output_html, $post_id );
138 |     }
139 |     return $output_html;
140 | }
141 |
142 | // get the total page views and daily page views for the post
143 | public static function pvc_stats_counter( $post_id, $increase_views = false ) {
144 |     global $wpdb;
            
```

Steps

Variables   Query Info   Search   Step Finder   Captures

- ✂ \$output\_html = (string) "12&nbsp;total views, 12&nbsp;views today"
- ✂ \$post\_id = (int) 26
- ★ \$results (stdClass) (2)
  - ✂ \$results->today = (string) "12"
  - ✂ \$results->total = (string) "12"
- ✂ \$total (uninitialized)
- ★ \$wpdb (wpdb) (60)
- ★ :: (A3\_PVC) (0)

Figure 4.31: Case Study 4: Differences Tool on Two Lines

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/pvc_class.php
130 |         $total = A3_PVC::pvc_fetch_post_total( $post_id );
131 |         if ( $total > 0 ) {
132 |             $output_html = number_format( $total ) . '&nbsp;'; __('total views', 'page-views-
count') . ', ' . __('no views today', 'page-views-count');
133 |         } else {
134 |             $output_html = __('No views yet', 'page-views-count');
135 |         }
136 |     }
137 |     $output_html = apply_filters( 'pvc_filter_get_stats', $output_html, $post_id );
138 | }
139 | return $output_html;
140 | }
141 |
142 | // get the total page views and daily page views for the post
143 | public static function pvc_stats_counter( $post_id, $increase_views = false ) {
144 |     global $wpdb;
145 |     global $pvc_settings;
146 |
147 |     $load_by_ajax_update_class = '';
148 |     if ( $increase_views ) $load_by_ajax_update_class = 'pvc_load_by_ajax_update';
149 |
150 |     // get the stats and
151 |     $html = "<div class='pvc_clear'></div>";
152 |
153 |     if ( $pvc_settings['enable_ajax_load'] == 'yes' ) {
154 |         $stats_html = "<p id='pvc_stats_'. $post_id. "' class='pvc_stats ".$load_by_ajax_update_class. "' elem
            
```

Steps

Variables   Query Info   Search   Step Finder   Captures

No Differences

zohbzhar.com/research/PEODit/PEODit.php?PCID=17

- (13\_73459) Line 128 of research/casestudies/wp/wpTheme/wp-c
- (13\_73508) Line 137 of research/casestudies/wp/wpTheme/wp-c
- (13\_73515) Line 139 of research/casestudies/wp/wpTheme/wp-c

**Figure 4.32:** Case Study 4: Results from `pvc_stats_counter()` of Home Page

The screenshot displays a debugger interface with two main panels. The left panel, titled 'Source Code', shows the PHP code for the `pvc_stats_counter()` function in `research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/pvc_class.php`. The code includes comments and logic for fetching page views, handling AJAX updates, and generating HTML for the stats. The right panel, titled 'Variables', shows the current state of variables during execution, including `$html`, `$increase_views`, `$load_by_ajax_update_class`, `$post_id`, `$pvc_settings`, `$stats_html`, `$wpdb`, and `::$(A3_PVC)`.

files, one could see that the two files retrieved the page content differently (see Figure 4.36). The home page used a `get_the_excerpt()` WordPress function to get the content and the page used a `the_content()` WordPress function. Looking in the documentation, these functions are associated with the `get_the_excerpt`<sup>14</sup> and `the_content`<sup>15</sup> hooks respectively. Curiously, it was noted that the plugin had a function for retrieving the page view count when an excerpt is requested (which is what is needed on the home page), but Execution Path Highlighting showed that this function wasn't being used for the home page (see Figure 4.37).

Using `grep`<sup>16</sup>, the plugin was searched for where these hooks were attached. Hooks are attached using the `add_action` and `add_filter` so these strings were used as search patterns. The hooks were found in a `plugin-init.php` file (see Figure 4.38). It appears that this is where the bug is. The plugin author decided to hook to `the_excerpt` and the theme was

<sup>14</sup>[https://developer.wordpress.org/reference/hooks/get\\_the\\_excerpt/](https://developer.wordpress.org/reference/hooks/get_the_excerpt/)

<sup>15</sup>[https://developer.wordpress.org/reference/hooks/the\\_content/](https://developer.wordpress.org/reference/hooks/the_content/)

<sup>16</sup>`grep` is a tool that can be used to search through files and content for a specific search pattern. See the manual page at <http://www.gnu.org/software/grep/manual/grep.html>

**Figure 4.33: Case Study 4: Results from pvc\_stats\_show() of Home Page**

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/pvc_class.php
243 |
244 | public static function pvc_stats_show($content){
245 |     remove_action('loop_end', array('A3_PVC', 'pvc_stats_echo'));
246 |     remove_action('genesis_after_post_content', array('A3_PVC', 'genesis_pvc_stats_echo'));
247 |     global $post;
248 |     if ( ! $post ) return;
249 |
250 |     $args=array(
251 |         'public' => true,
252 |         '_builtin' => false
253 |     );
254 |     $output = 'names'; // names or objects, note names is the default
255 |     $operator = 'and'; // 'and' or 'or'
256 |     $post_types = get_post_types($args, $output, $operator );
257 |
258 |     global $pvc_settings;
259 |     if ( empty( $pvc_settings ) ) {
260 |         $pvc_settings = get_option('pvc_settings', array() );
261 |     }
262 |
263 |     if ( self::pvc_is_activated( $post->ID ) ) {
264 |         if ( ! is_singular() || is_singular( $post_types ) ) {
265 |             if ( ! isset( $pvc_settings['enable_ajax_load'] ) || $pvc_settings['enable_ajax_load'] != 'yes' ) {
266 |                 A3_PVC::pvc_stats_update($post->ID);
267 |             }
268 |             $content .= A3_PVC::pvc_stats_counter($post->ID, true );
269 |         } else {
270 |             $content .= A3_PVC::pvc_stats_counter($post->ID);
271 |         }
272 |     }
273 |     return $content;
274 | }
                
```

Steps ⬆ ⬅ ➡ ⬆

Variables Query Info Search Step Finder Captures

- ▶ \$args (array) (2)
  - ✂ \$content = (string) "This is the homepage of a test WP site.<div class="pvc\_clear"></div>"
  - ✂ \$operator = (string) "and"
  - ✂ \$output = (string) "names"
- ▶ ★ \$post (WP\_Post) (24)
  - ✂ \$post\_types (array) (0)
  - ✂ \$pvc\_settings (array) (3)
  - ★ :: (A3\_PVC) (0)

**Figure 4.34: Case Study 4: Using Step Finder**

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/pvc_class.php
243 |
244 | public static function pvc_stats_show($content){
245 |     remove_action('loop_end', array('A3_PVC', 'pvc_stats_echo'));
246 |     remove_action('genesis_after_post_content', array('A3_PVC', 'genesis_pvc_stats_echo'));
247 |     global $post;
248 |     if ( ! $post ) return;
249 |
250 |     $args=array(
251 |         'public' => true,
252 |         '_builtin' => false
253 |     );
254 |     $output = 'names'; // names or objects, note names is the default
255 |     $operator = 'and'; // 'and' or 'or'
256 |     $post_types = get_post_types($args, $output, $operator );
257 |
258 |     global $pvc_settings;
259 |     if ( empty( $pvc_settings ) ) {
260 |         $pvc_settings = get_option('pvc_settings', array() );
261 |     }
262 |
263 |     if ( self::pvc_is_activated( $post->ID ) ) {
264 |         if ( ! is_singular() || is_singular( $post_types ) ) {
265 |             if ( ! isset( $pvc_settings['enable_ajax_load'] ) || $pvc_settings['enable_ajax_load'] != 'yes' ) {
266 |                 A3_PVC::pvc_stats_update($post->ID);
267 |             }
268 |             $content .= A3_PVC::pvc_stats_counter($post->ID, true );
269 |         } else {
270 |             $content .= A3_PVC::pvc_stats_counter($post->ID);
271 |         }
272 |     }
273 |     return $content;
274 | }
                
```

Steps ⬆ ⬅ ➡ ⬆

Variables Query Info Search Step Finder Captures

Find All Steps at Line Number

Line Number (Optional)

File

Find Steps

Results:

Step: 164356



Figure 4.35: Case Study 4: Results from `pvc_stats_show()` of Page

Source Code

```

research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/pvc_class.php
242 | }
243 |
244 | public static function pvc_stats_show($content){
245 |     remove_action('loop_end', array('A3_PVC', 'pvc_stats_echo'));
246 |     remove_action('genesis_after_post_content', array('A3_PVC', 'genesis_pvc_stats_echo'));
247 |     global $post;
248 |     if ( ! $post ) return;
249 |
250 |     $args=array(
251 |         'public' => true,
252 |         'builtin' => false
253 |     );
254 |     $output = 'names'; // names or objects, note names is the default
255 |     $operator = 'and'; // 'and' or 'or'
256 |     $post_types = get_post_types($args, $output, $operator );
257 |
258 |     global $pvc_settings;
259 |     if ( empty( $pvc_settings ) ) {
260 |         $pvc_settings = get_option('pvc_settings', array() );
261 |     }
262 |
263 |     if ( self::pvc_is_activated( $post->ID ) ) {
264 |         if ( is_singular() || is_singular( $post_types ) ) {
265 |             if ( ! isset( $pvc_settings['enable_ajax_load'] ) || $pvc_settings['enable_ajax_load'] != 'yes' ) {
266 |                 A3_PVC::pvc_stats_update($post->ID);
267 |             }
268 |             $content .= A3_PVC::pvc_stats_counter($post->ID, true );
269 |         } else {
270 |             $content .= A3_PVC::pvc_stats_counter($post->ID);
271 |         }
272 |     }
273 |     return $content;
274 | }
                
```

Steps

Variables Query Info Search Step Finder Captures

```

$args (array) (2)
  $content = (string) "This is a test count page.<div class="pvc_clear"></div><p class="pvc
  $operator = (string) "and"
  $output = (string) "names"
  $post (WP_Post) (24)
  $post_types (array) (0)
  $pvc_settings (array) (3)
  * :: (A3_PVC) (0)
                
```

Figure 4.36: Case Study 4: How the Home Page versus the Test Page Retrieves Content in the Theme

research/casestudies/wp/wpTheme/wp-content/themes/trident-lite/front-page.php

```

121 | </h6>
122 |
123 |     <div class="entry-meta">
124 |         <?php trident_posted_on(); ?>
125 |     </div><!-- .entry-meta -->
126 | </header><!-- .entry-header -->
127 |
128 |     <div class="entry-content">
129 |         <?php echo trident_shorten_excerpt( get_the_excerpt() ); ?>
130 |     </div><!-- .entry-content -->
131 | </div>
132 |
133 | <?php endwhile; ?>
134 |
135 |     <div class="clearfix"></div>
136 |
137 |     <?php trident_paging_nav(); ?>
138 |
139 | <?php else : ?>
140 |
141 |     <?php get_template_part( 'content', 'none' ); ?>
142 |
143 | <?php endif; ?>
144 |
145 | </main><!-- #main -->
146 | </div><!-- #primary -->
                
```

Steps

research/casestudies/wp/wpTheme/wp-content/themes/trident-lite/content-page.php

```

01 | <?php
02 | /**
03 |  * The template used for displaying page content in page.php
04 |  *
05 |  * @package trident
06 |  */
07 | ?>
08 |
09 | <article id="post-?php the_ID(); ?>" <?php post_class(); ?>
10 | <header class="entry-header">
11 |     <h1 class="entry-title"><?php the_title(); ?></h1>
12 | </header><!-- .entry-header -->
13 |
14 |     <div class="entry-content">
15 |         <?php the_content(); ?>
16 |     </div>
17 |     <?php
18 |         wp_link_pages( array(
19 |             'before' => '<div class="page-links">'. __( 'Pages:', 'trident' ),
20 |             'after' => '</div>',
21 |         ) );
22 |     </div><!-- .entry-content -->
23 |     <?php edit_post_link( __( 'Edit', 'trident' ), '<footer class="entry-meta">
24 | </article><!-- #post-# -->
                
```

Steps

**Figure 4.37:** Case Study 4: Using the Normal Version Instead of Excerpt Version on the Home Page

```
research/casestudies/wp/wpTheme/wp-content/plugins/page-views-  
count/pvc_class.php ▾  
243 |  
244 | public static function pvc_stats_show($content){  
245 |     remove_action('loop_end', array('A3_PVC', 'pvc_stats_echo'));  
246 |     remove_action('genesis_after_post_content', array('A3_PVC', 'genesis_pvc_stats_echo'));  
247 |     global $post;  
248 |     if ( ! $post ) return;  
249 |  
250 |     $args=array(  
251 |         'public' => true,  
252 |         '_builtin' => false  
253 |     );  
254 |     $output = 'names'; // names or objects, note names is the default  
255 |     $operator = 'and'; // 'and' or 'or'  
256 |     $post_types = get_post_types($args, $output, $operator );  
257 |  
258 |     global $pvc_settings;  
259 |     if ( empty( $pvc_settings ) ) {  
260 |         $pvc_settings = get_option('pvc_settings', array() );  
261 |     }  
262 |  
263 |     if ( self::pvc_is_activated( $post->ID ) ) {  
264 |         if ( is_singular() || is_singular( $post_types ) ) {  
265 |             if ( ! isset( $pvc_settings['enable_ajax_load'] ) || $pvc_settings['enable_ajax_load'] != 'yes' ) {  
266 |                 A3_PVC::pvc_stats_update($post->ID);  
267 |             }  
268 |             $content .= A3_PVC::pvc_stats_counter($post->ID, true );  
269 |         } else {  
270 |             $content .= A3_PVC::pvc_stats_counter($post->ID);  
271 |         }  
272 |     }  
273 |     return $content;  
274 | }  
275 |  
276 | public static function excerpt_pvc_stats_show($excerpt){  
277 |     remove_action('loop_end', array('A3_PVC', 'pvc_stats_echo'));  
278 |     remove_action('genesis_after_post_content', array('A3_PVC', 'genesis_pvc_stats_echo'));
```



Used

Not Used

**Figure 4.38:** Case Study 4: Initialization of Hooks

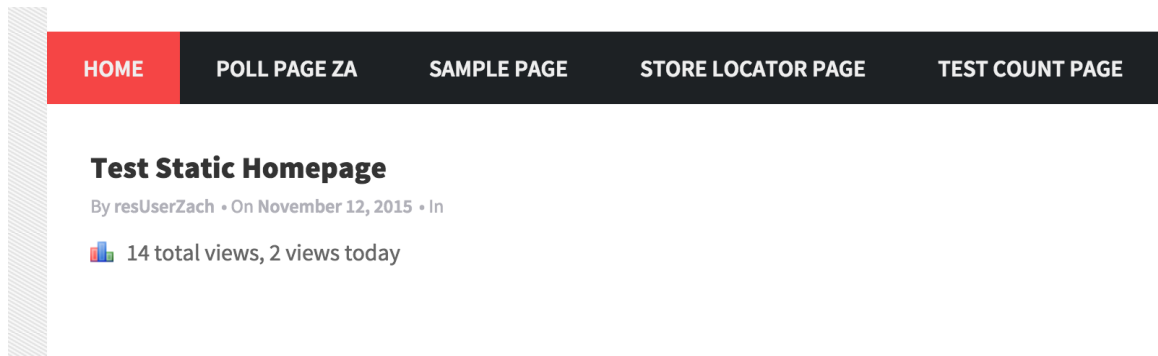
```
research/casestudies/wp/wpTheme/wp-content/plugins/page-views-count/admin/plugin-  
init.php ▾  
083 |     global $wpdb;  
084 |     $wpdb->query("DELETE FROM " . $wpdb->prefix . "pvc_daily WHERE time <= '".date('Y-  
m-d', strtotime('-2 days'))."");  
085 | }  
086 |  
087 | add_action('genesis_after_post_content', array('A3_PVC', 'genesis_pvc_stats_echo'));  
088 | //add_action('loop_end', array('A3_PVC', 'pvc_stats_echo'), 9);  
089 | add_filter('the_content', array('A3_PVC', 'pvc_stats_show'), 8);  
090 | //add_filter('the_excerpt', array('A3_PVC', 'excerpt_pvc_stats_show'), 8);  
091 | add_filter('get_the_excerpt', array('A3_PVC', 'excerpt_pvc_stats_show'), 8);  
092 |
```

using the *get\_the\_excerpt*. Interestingly, the author had a line to hook to the *get\_the\_excerpt* event but it was commented out (also visible in Figure 4.38). To attempt to fix the bug, the comment was removed from this line and the faulty hook was commented out. Now when visiting the homepage, it appeared that the bug was fixed and the page count was displaying correctly (see Figure 4.39).

#### 4.4.5 Analysis

As the user is presumed to be new to WordPress development, it cannot be said if the bug was caused by the theme's author using the wrong hook or the plugin's author. However, what can be said is that PECCit made it easy to navigate through the framework and find what was needed to fix the bug. This case study showed the use of variable tracing, variable inspection, variable differencing, the Step Finder, the Search tool, Execution Path Highlighting, and backward step operations. Using the Search tool, one was able to quickly

**Figure 4.39:** *Case Study 4: Home Page with Correct Page Count Formatting*



jump into the action without having to scan through source code and guess where to start debugging.

With a standard debugger, the user would have had to guess where to set a breakpoint and hope that they didn't over jump. Also, the user most likely couldn't have had two debugging sessions going on at once with a traditional debugger so they would need to write down or remember the variable names to compare. With this example and PECCit, the user had the two sessions open in different tabs and easily looked between them (see Figure 4.36). Variable inspection was used to see where variables were created/changed and the user jumped back in time to when the page view stats were calculated. Variable differencing was used to see if certain lines changed the variables that appeared to be related to the bug. The user was able to step out and back to understand the call stack (but then step in if needed and move through time freely). The traces required around 6 minutes each which, understandably, most developers would probably not want to wait. Thus, further improvements to speed could make this case study even more realistic (see Section 5.1.1).

## 4.5 Case Study 5: Capitalized Titles in Drupal Theme

### 4.5.1 Background

Similarly to WordPress, Drupal has a main core (which handles content management, users, etc), modules (which add functionality like WordPress plugins), and a theme (which is responsible for the appearance of the site).

### 4.5.2 Problem

Using the Business theme<sup>17</sup>, a user was finding that the theme was automatically capitalizing all of the words in the titles of their content (see Figure 4.40 for the forum post).<sup>18</sup> They preferred that the theme leave the capitalization decisions to the user as they did not want words like “the” and “a” to be capitalized.<sup>19</sup>

### 4.5.3 Setup

With the assumption that the developer is new to Drupal (and perhaps even web development), a reasonable guess as to what was happening is that some code in their Drupal instance (whether it is Drupal core or the theme) was overriding their title content with capitalization. To see this in action, Drupal was installed on a research server and the Business theme was downloaded and installed. An article was created with the title “all lowercase

---

<sup>17</sup><https://www.drupal.org/project/business>

<sup>18</sup><https://www.drupal.org/node/2579699>

<sup>19</sup>Assuming that the theme is causing this to happen, it is most likely a *feature* of the theme instead of a *bug*. However, this case study refers to this action as a *bug* since the code causing the capitalization is incorrectly displaying the content (according to the users and their desired behavior).

**Figure 4.40: Case Study 5: Forum Post**

[Business](#) » [Issues](#)

## Capitalization in Titles

Hello there,

We are seeing that the system is automatically capitalizing words in the Title fields of items. This is not desired as such words as "the", "a" and "to" should be lower case.

How can this be overridden so that the theme leaves the capitalization decisions to the user?

Thank you.

Denis.

### Comments

[denisgre](#) created an issue. #1

[Log in](#) or [register](#) to post comments

Active	
<b>Project:</b>	Business
<b>Version:</b>	7.x-1.11
<b>Component:</b>	Code
<b>Priority:</b>	Normal
<b>Category:</b>	Bug report
<b>Assigned:</b>	Unassigned
<b>Reporter:</b>	<a href="#">denisgre</a>
<b>Created:</b>	October 3, 2015 - 18:34
<b>Updated:</b>	October 3, 2015 - 18:34

[Log in](#) or [register](#) to update this issue

Jump to:  
[Most recent comment](#)

title please” with specifically all lowercase characters. When the front page was visited, the bug was recreated successfully where the title was capitalized (see Figure 4.41).


#### 4.5.4 Using PECCit


It could be beneficial to see where in the execution the title was getting printed. Perhaps the theme was overriding the title string with a function like `strtoupper()`<sup>20</sup>. From here, there are two initial strategies that could be used. One, the developer could search through all of the source code for that function and all similar functions (like `ucfirst()`<sup>21</sup>) using a tool like `grep` which could yield nothing. Two, the user could trace an execution and use the Search Tool in the PECCit Inspector to look for a `title` variable (assuming, possibly

<sup>20</sup><http://php.net/manual/en/function.strtoupper.php>

<sup>21</sup><http://php.net/manual/en/function.ucfirst.php>

**Figure 4.41:** *Case Study 5: Capitalized Content*

 **Drupal Case Study Business** [Home](#) all lowercase title please



**What We Do**

**All Lowercase Title Please**

Submitted by resUserZach on Thu, 10/08/2015 - 11:56

This is a test post to see if the title gets capitalized

[Read more](#) [Log in](#) or [register](#) to post comments

**User Login**

**Username \***

**Password \***

erroneously, that there is one with that name) or for the value “all lowercase title please” (assuming, again possibly erroneously, that the title is stored perfectly in that format and a variable takes on that value). These strategies seemed error prone and time consuming. Instead, the capture tool was tried first to solve the issue with the goal of watching the page as it’s being built.



When capturing, strict settings are important to reduce execution overhead so it is worth taking the time to think about the settings before tracing. Since it was assumed the developer understands that Drupal uses themes for appearance, though they don’t need to know how they work, a good first try was to whitelist the entire theme. Since whitelisting an entire folder could be expensive, the settings for variable tracing (Property Max Depth and Property Max Children) were set very conservatively. Also since it can’t be assumed at what depth the theme would execute, a deep Step Max Depth was used. With the settings adjusted using the PECCit Session Manager, the homepage was visited with capturing enabled.




The session only required 25 seconds to trace (see Figure 4.42). When examined with the PECCit Inspector, it was not immediately obvious when/where in the execution the title was printed. Since the theme is responsible for printing, it was most likely printed in a theme file but finding the execution would have taken a long time using the Step Forward and Step In tools. Thus, the user decided they wanted to view the captures in one of the theme files. As there were multiple files, the first file tried was *html.tpl.php* (see Figure 4.43) and the first step in the file was found using the Step Finder and clicked.

With the Capture Pane open, the Step Forward button was repeatedly pressed. The file appeared to be setting up the HTML with the basic necessities like *<head >* information.



**Figure 4.42: Case Study 5: The PECCit Session**

Sessions  





SID	Query	Timestamp	Status	Steps	Time	Select Time	Insert Time	Messages	Inserts	DB Incr (MB)	Settings	Error Msg	Actions
1	/research/casestudies/drupalBu...	12/18/15 3:53:47 PM	Done Collecting	29921	00:00:25	00:00:04	00:00:16	62595	58355	8.42188	Capture_On : true;Exclude Address : true;Property Max Children : 5;Property Max Depth : 1;Site : http://www.zachazar.com;Step Max Depth : 50;Whitelist : themes/business;		  

**Figure 4.43: Case Study 5: Choosing the File in Step Finder**

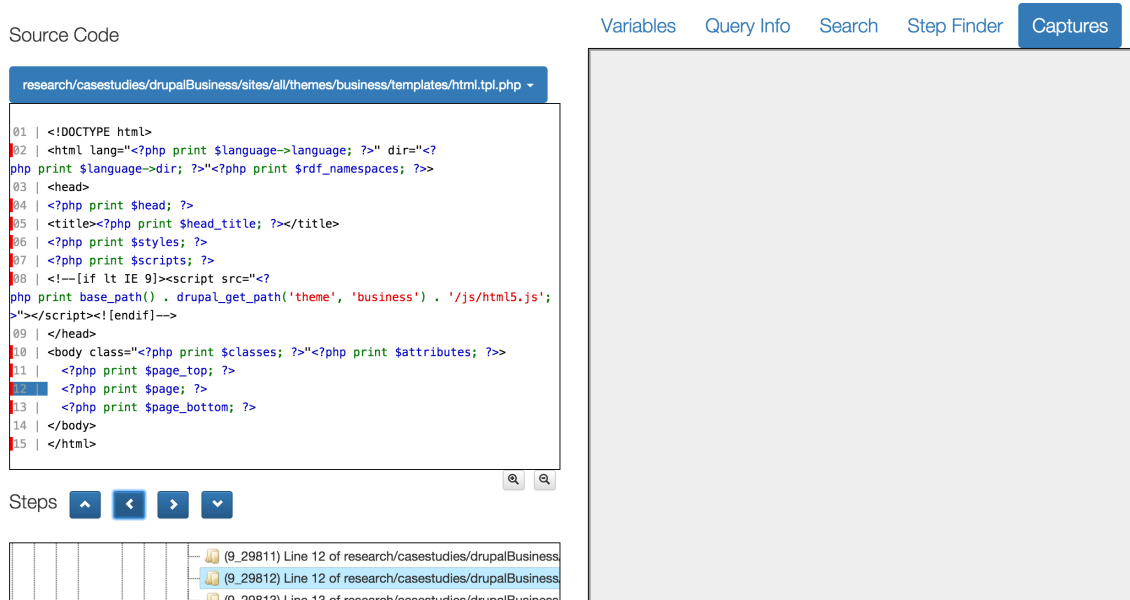
Source Code

Source Files ▾

- research/casestudies/drupalBusiness/modules/path/path.module
- research/casestudies/drupalBusiness/modules/rdi/rdi.module
- research/casestudies/drupalBusiness/modules/search/search.module
- research/casestudies/drupalBusiness/modules/shortcut/shortcut.module
- research/casestudies/drupalBusiness/modules/system/system.module
- research/casestudies/drupalBusiness/modules/taxonomy/taxonomy.module
- research/casestudies/drupalBusiness/modules/toolbar/toolbar.module
- research/casestudies/drupalBusiness/modules/user/user-picture.tpl.php
- research/casestudies/drupalBusiness/modules/user/user.module
- research/casestudies/drupalBusiness/profiles/standard/standard.profile
- research/casestudies/drupalBusiness/sites/all/themes/business/template.php
- research/casestudies/drupalBusiness/sites/all/themes/business/templates/block.tpl.php
- ✓ research/casestudies/drupalBusiness/sites/all/themes/business/templates/html.tpl.php**
- research/casestudies/drupalBusiness/sites/all/themes/business/templates/node.tpl.php
- research/casestudies/drupalBusiness/sites/all/themes/business/templates/page--front.tpl.php
- research/casestudies/drupalBusiness/sites/all/themes/business/templates/region.tpl.php
- research/casestudies/drupalBusiness/sites/default/settings.php
- research/casestudies/drupalBusiness/themes/engines/phptemplate/phptemplate.engine

Steps    

**Figure 4.44:** Case Study 5: Before Page Print



During these steps, the Captures were blank. Then when line 12 was executed, the entire page was printed with the capitalization error (see Figures 4.44 and 4.45). Looking at the *\$page* variable in the Variable Pane, it appeared the page had already been built and was just being printed at this line (see Figure 4.46). Perhaps another theme file was responsible for building this variable and it was overriding the title text. Next, the *page-front.tpl.php* was selected in the Step Finder.

Using the same strategy, the Step Forward button was pressed continuously while watching the Capture Pane. The file appeared to be constructing the page and (interestingly) the title of the article was lowercase. Also, the page appeared to be unstyled (see Figures 4.47 and 4.48). The page content was identical to the styled page and it seemed that the only thing that was different was the styling and the capitalization.

Figure 4.45: Case Study 5: After Page Print

Source Code

```


research/casestudies/drupalBusiness/sites/all/themes/business/templates/html.tpl.php
01 | <!DOCTYPE html>
02 | <html lang="<?php print $language->language; ?>" dir="<?
php print $language->dir; ?>"><?php print $rdf_namespaces; ?>
03 | <head>
04 | <?php print $head; ?>
05 | <title><?php print $head_title; ?></title>
06 | <?php print $styles; ?>
07 | <?php print $scripts; ?>
08 | <!--[if lt IE 9]><script src="<?
php print base_path() . drupal_get_path('theme', 'business') . '/js/html5.js';
>"></script><![endif]-->
09 | </head>
10 | <body class="<?php print $classes; ?>"><?php print $attributes; ?>
11 | <?php print $page_top; ?>
12 | <?php print $page; ?>
13 | <?php print $page_bottom; ?>
14 | </body>
15 | </html>

```

Steps

Variables Query Info Search Step Finder Captures

**Drupal Case Study Business**



What We Do

All Lowercase Title Please

Submitted by resUserZach on Thu, 10/08/2015 - 11:56

Figure 4.46: Case Study 5: \$page Variable

Source Code

```

research/casestudies/drupalBusiness/sites/all/themes/business/templates/html.tpl.php
01 | <!DOCTYPE html>
02 | <html lang="<?php print $language->language; ?>" dir="<?
php print $language->dir; ?>"><?php print $rdf_namespaces; ?>
03 | <head>
04 | <?php print $head; ?>
05 | <title><?php print $head_title; ?></title>
06 | <?php print $styles; ?>
07 | <?php print $scripts; ?>
08 | <!--[if lt IE 9]><script src="<?
php print base_path() . drupal_get_path('theme', 'business') . '/js/html5.js';
>"></script><![endif]-->
09 | </head>
10 | <body class="<?php print $classes; ?>"><?php print $attributes; ?>
11 | <?php print $page_top; ?>
12 | <?php print $page; ?>
13 | <?php print $page_bottom; ?>
14 | </body>
15 | </html>

```

Steps

Variables Query Info Search Step Finder Captures

- \$attributes = (string) ""
- \$attributes\_array = (array) (0)
- \$classes = (string) "html front not-logged-in one-sidebar sidebar-first page-node"
- \$classes\_array = (array) (5)
- \$content\_attributes = (string) ""
- \$content\_attributes\_array = (array) (0)
- \$css = (array) (11)
- \$db\_is\_active = (bool) True
- \$directory = (string) "sites/all/themes/business"
- \$grddl\_profile = (string) "http://www.w3.org/1999/xhtml/vocab"
- \$head = (string) "<meta charset='utf-8' /> <link rel='shortcut icon' href='http://zachaz"
- \$head\_title = (string) "Drupal Case Study Business"
- \$head\_title\_array = (array) (1)
- \$id = (int) 1
- \$is\_admin = (bool) False
- \$is\_front = (bool) True
- \$language (stdClass) (12)
- \$logged\_in = (bool) False
- \$page = (string) " <div id='wrap'> <header id='header' class='clearfix' role='banner'>**
- \$page\_bottom = (string) ""
- \$page\_top = (string) ""
- \$rdf\_namespaces = (string) \* xmlns:content="http://purl.org/rss/1.0/modules/content/"

Figure 4.47: Case Study 5: First Capture of Unstyled Front Page

Source Code

```

research/casestudies/drupalBusiness/sites/all/themes/business/templates/page--
front.tpl.php
995 | </header>
996 |
997 | <?php print render($page['header']); ?>
998 |
999 | <?php if (theme_get_setting('slideshow_display','business')): ?>
1000 | <?php
1001 | $url1 = check_plain(theme_get_setting('slide1_url','business'));
1002 | $url2 = check_plain(theme_get_setting('slide2_url','business'));
1003 | $url3 = check_plain(theme_get_setting('slide3_url','business'));
1004 | ?>
1005 | <div id="slider">
1006 |   <div class="main_view">
1007 |     <div class="window">
1008 |       <div class="image_reel">
1009 |         <a href="<?php print url($url1); ?>"></a>
1010 |         <a href="<?php print url($url2); ?>"></a>
1011 |         <a href="<?php print url($url3); ?>"></a>
1012 |         ...

```

Steps ▲ ◀ ▶ ▼

Variables Query Info Search Step Finder Captures

### Drupal Case Study Business

- [Home](#)
- [all lowercase title please](#)

Figure 4.48: Case Study 5: Second Capture of Unstyled Front Page

Source Code


```

research/casestudies/drupalBusiness/sites/all/themes/business/templates/page--
front.tpl.php
170 | <?php endif; ?>
171 |
172 | <?
173 | php if ($page['footer_first'] || $page['footer_second'] || $page['footer_third'] || $page['f
174 | >
175 | <div id="footer-saran" class="clearfix">
176 |   <div id="footer-wrap">
177 |     <?php if ($page['footer_first']): ?>
178 |     <div class="footer-box"><?php print render($page['footer_first']); ?></div>
179 |     <?php endif; ?>
180 |     <?php if ($page['footer_second']): ?>
181 |     <div class="footer-box"><?php print render($page['footer_second']); ?></div>
182 |     <?php endif; ?>
183 |     <?php if ($page['footer_third']): ?>
184 |     <div class="footer-box"><?php print render($page['footer_third']); ?></div>
185 |     <?php endif; ?>
186 |     <?php if ($page['footer_fourth']): ?>
187 |     <div class="footer-box remove-margin"><?
php print render($page['footer_fourth']); ?></div>
188 |     <?php endif; ?>
189 |   </div>
190 | </div>
191 | <div class="clear"></div>

```

Steps ▲ ◀ ▶ ▼

Variables Query Info Search Step Finder Captures



1 2 3

### all lowercase title please

Submitted by resUserZach on Thu, 10/08/2015 - 11:56

This is a test post to see if the title gets capitalized

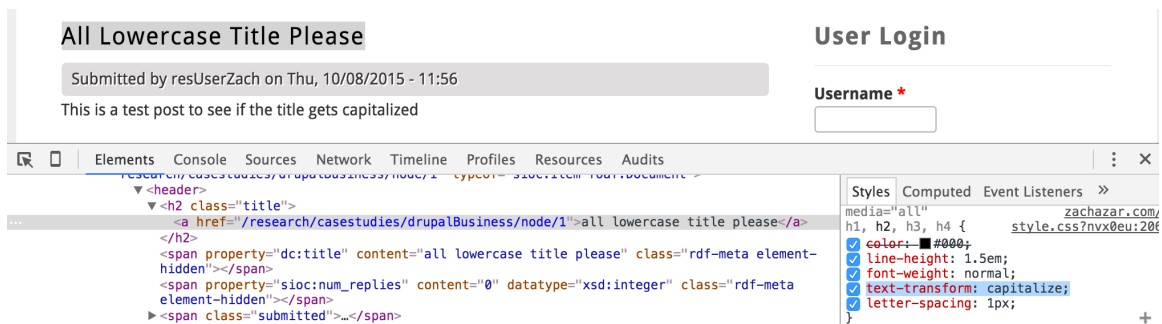
- [Read more about all lowercase title please](#)
- [Log in](#) or [register](#) to post comments

### User login

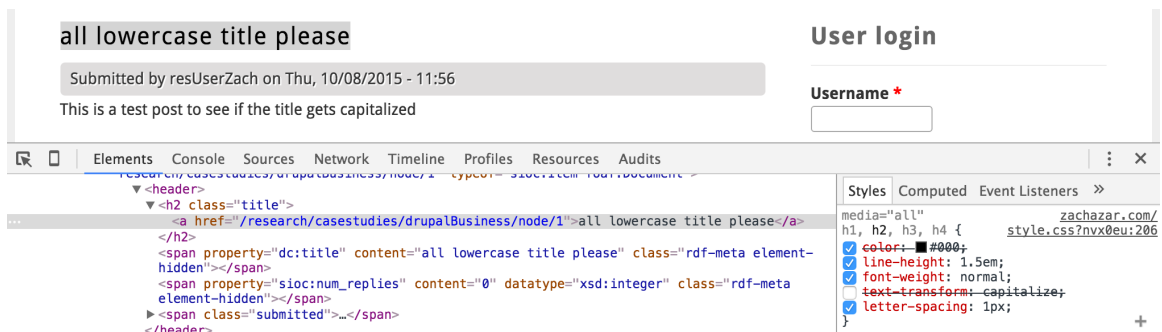
Username \*

Password \*

**Figure 4.49:** Case Study 5: Chrome DevTools Showing CSS Properties



**Figure 4.50:** Case Study 5: Chrome DevTools With Deselected CSS Property



With this discovery, the user hypothesized that the culprit was actually CSS and not a backend framework issue. In Google Chrome, the title link on the homepage was inspected using the Google Chrome DevTools<sup>22</sup> to see the CSS properties for the element. Here, the bug was found. The `<h2>` element had a `text-transform` CSS property set to `capitalize` which was most likely causing the capitalization (see Figure 4.49). To test, the property was deselected in Chrome DevTools which resulted in the title being displayed as it was intended with all lowercase letters (see Figure 4.50).

<sup>22</sup><https://developer.chrome.com/devtools>

### **4.5.5 Analysis**

Though the bug was ultimately caused by a front end issue with CSS, PECCit was still helpful in identifying the problem. The user was able to smartly trace the execution with capturing enabled in under half a minute. The user could see the various theme files and how they played a role in constructing the page. With capturing, it's very easy to get mesmerized stepping through the execution watching as the preview changes line-by-line. As an educational tool, it enables the user to learn more about a framework without telling them but by showing them and giving them the tools to explore the framework themselves. As a debug tool, the capture functionality allows the developer to quickly narrow down files and functions where the bug could be as they watch how these functions interact with page construction.

## **4.6 Case Study Analysis**

While these case studies do not prove the PECCit system's performance or novelty, they demonstrate its usefulness when debugging real-world web development problems. In these case studies, the user was able to get a better grasp of the internal workings of the target framework while quickly moving through the execution to find/understand the bugs that were presented. As with all PECCit traces, the debugging sessions themselves could be saved and shared with other users. The case studies demonstrate that novice users could use the tool to learn more about a web framework (educationally helpful) and developers could use the tool to quickly examine source code, variable histories, the execution path, and page creation to find and fix bugs.

These case studies indirectly present some of the shortcomings of the PECCit system as well. First, variable tracing can be slow and resource consuming. The storage of lots of variables in a large framework like WordPress can quickly grow in size on the server. Large traces also require quite a bit of time, which many users might not readily adopt even if the trace could help find the bug. Since these traces can grow so quickly, the proper tweaking of settings and whitelisting is crucial. This means that using PECCit could have a learning curve as it takes some practice to recognize which files should be whitelisted and what settings will likely maintain a balance between performance and adequate information needed to find a bug.

The PECCit system offers powerful tools, demonstrated in these case studies, that could make PECCit more effective than traditional debuggers for web development. In these case studies, the Step Forward and Backward functionalities were utilized to move forward and backward in time. This is quite powerful compared to the linear forward movement of a traditional debugger. The Search functionality was used to quickly find variables and values of interest (something that traditional debuggers would be unable to do since variable values are not known after an execution). The Execution Path Highlighting feature was demonstrated allowing the user to quickly see which lines of code were executed. Traditional debuggers are stuck in a moment in time and (besides knowing the call stack) do not know which functions and files are going to be used or which were referenced in the past. The Step Finder was regularly used to quickly jump to a line of code and a time in execution (quick arbitrary access).

The Variable Inspector was utilized to look back through a variable's history to see exactly when the variable took on certain values. Traditional debuggers do not save this infor-

mation. Some traditional debuggers offer conditional breakpoints, like “break when  $x=5$ ” but this breakpoint needs to be very specific and one must think of these breakpoints before running the execution. It was demonstrated that the sessions are automatically recorded and saved so unexpected requests (like an AJAX call) are not missed. Also, these sessions can be inspected in parallel and shared with others. Finally, PECCit offers capturing which can let the user see the page as it’s being built. The case study utilizing capturing demonstrated how the tool can be used educationally as well as for debugging. It demonstrated that capturing is even helpful when debugging CSS on the frontend by showing how it alters the display of the content.



# Chapter 5

## Conclusion

Debugging can be an extremely costly, but important, task for a developer. The time required to rid bugs from a system can take weeks if not months and thousands of dollars [Gib94]. Surprisingly, debugging strategies commonly used by developers are outdated and newer debugging technologies are not widely accepted [SPTH14]. If more developers and programmers used back-in-time debuggers, they could potentially save a lot of time and money.

Omniscient debugging is a back-in-time debugging strategy which allows the user to trace an execution and examine it. The user can step forward and backward through the execution, examine variable histories, and instantly access arbitrary steps in the execution. As debugging often requires developers to travel backward through infected states to find the initial defect, omniscient debuggers can greatly assist developers with their back-in-time nature. Other debugging strategies exist as well like replay-based debugging, query-

based debugging, and reverse-executing debuggers which also offer much more features than the standard breakpoint debugger.

Presented along with this thesis is PECCit, an implementation of an omniscient debugger for backend web developers. PECCit is able to trace a PHP framework and display the execution information (control and data flow) to the user in a browser based IDE. With PECCit, the user can navigate forward and backward in time, examine variable histories, inspect the execution path, and search through program states. PECCit even offers a novel feature called capturing allowing the user to watch as the web page is built line-by-line. The system was designed to be language independent such that future work could extend the tool to other languages. PECCit is a powerful tool that developers can use to debug programs more efficiently using back-in-time strategies. PECCit's various features were demonstrated in case studies of real-world problems and compared to tactics used with standard debuggers.

## **5.1 Future Work**

PECCit is a distributed system with lots of moving pieces. It can always be improved upon. These improvements could make the system faster and traces smaller. They could provide more features to developers using the tool. They could also make the tool more language independent allowing for a larger user base. The following sections examine these improvements.

### **5.1.1 Performance Improvements**

Though PECCit offers reasonable performance, as shown in the case studies in Chapter 4, the current implementation was designed with prototype and research proof-of-concept in mind rather than performance. To decrease execution overhead, multithreading could be used by the ADS when parsing and handling messages. Also to reduce overhead, other database techniques could be experimented with like in-memory replicas and index improvements. Additional filtering options (filter by function, file type, lines of code, etc.) could help the user make the settings more specific to improve performance as well. Also, static analysis of the code could provide a major boost to performance. By examining the source code prior to execution or on-the-fly, PECCit could perhaps make assertions about which variables could change on each line resulting in fewer messages needed to ask for variables.

### **5.1.2 Additional Features**

PECCit could also be improved with additional features. One feature would be to snapshot the source code files between sessions to guarantee that the debug sessions correctly match up with the source code even if the source code has changed. Since PECCit is aware of the entire control and data flow, other features for the PECCit Inspector could be implemented to better help the developer understand the system (like control flow diagrams similar to those used in Diver [MS10] and data flow diagrams like those used in JIVE [RR05]). Other features could be added to the PECCit Inspector to improve functionality like the ability to annotate and save information in a session, to bookmark or save step lo-

cations, improved lazy loading of steps to decrease load time, and an improved appearance with more cross-browser compatibility.

### **5.1.3 Language Independence Improvements**

Though the structural design of PECCit encourages language independence by using remote debugging instead of code injection, there is still work to be done to allow PECCit to be truly language independent. First, PECCit needs to be tested extensively with other languages. Most likely, the ADS would need to be improved to account for small language differences. Also, capturing currently relies on a language feature (specifically PHP's Output Buffering). Similar utilities would need to be explored for other languages. As mentioned in Section 3.5.1.6, other languages offer similar features to output buffering that could be used to implement capturing. In future versions of PECCit, the ADS would detect automatically (from the debug engine) which language is being executed and would switch its capturing strategy to reflect the language. Further work is necessary to implement and test these language independent features.

# Bibliography

- [ACS84] James E. Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst.*, 6(1):1–19, January 1984.
- [ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Softw.*, 8(3):21–26, May 1991.
- [ADTP10] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 265–274, New York, NY, USA, 2010. ACM.
- [AKD<sup>+</sup>08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08*, pages 261–272, New York, NY, USA, 2008. ACM.
- [AKD<sup>+</sup>10] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *Software Engineering, IEEE Transactions on*, 36(4):474–494, July 2010.
- [Bal69] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference, AFIPS '69 (Spring)*, pages 567–580, New York, NY, USA, 1969. ACM.
- [Bal08] Melinda-Carol Ballou. Improving software quality to drive business agility. *IDC Survey and White Paper*, 2008.
- [BBKE13] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th An-*

*nual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 473–484, New York, NY, USA, 2013. ACM.

- [BCdJ<sup>+</sup>06] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.
- [BM14] Earl T. Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. *SIGPLAN Not.*, 49(10):67–82, October 2014.
- [Boo00] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 299–310, New York, NY, USA, 2000. ACM.
- [CFC01] Shyh-Kwei Chen, W. Kent Fuchs, and Jen-Yao Chung. Reversible debugging using program instrumentation. *IEEE Trans. Softw. Eng.*, 27(8):715–727, August 2001.
- [CJ07] Jeffrey K. Cxyz and Bharat Jayaraman. Declarative and visual debugging in eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 31–35, New York, NY, USA, 2007. ACM.
- [CS98] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multi-threaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.
- [CZ05] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.
- [DPS96] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA, 1996. ACM.
- [Duc98] Mireille Ducassé. Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events. Research Report RR-3489, INRIA, 1998.

- [Duc99a] Mireille Ducassé. Coca: An automated debugger for c. In *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pages 504–513, New York, NY, USA, 1999. ACM.
- [Duc99b] Mireille Ducassé. Opium: An extendable trace analyzer for prolog. *The Journal of Logic programming*, 39(1):177–223, 1999.
- [EAW10] Jakob Engblom, Daniel Aarno, and Bengt Werner. Full-system simulation from embedded to high-performance systems. In Rainer Leupers and Olivier Temam, editors, *Processor and System-on-Chip Simulation*, pages 25–45. Springer US, 2010.
- [Eis97] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, April 1997.
- [Eng12] J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–6, Sept 2012.
- [FB88] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, PADD '88*, pages 112–123, New York, NY, USA, 1988. ACM.
- [GDJ02] Y. Guéhéneuc, R. Douence, and N. Jussien. No java without caffeine: A tool for dynamic analysis of java programs. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 117–126, 2002.
- [GH93] Michael Golan and David R Hanson. Duel-a very high-level debugging language. In *USENIX Winter*, volume 107, page 118. Citeseer, 1993.
- [Gib94] W Wayt Gibbs. Software’s chronic crisis. *Scientific American*, 271(3):72–81, 1994.
- [GJ04] Paul V. Gestwicki and Bharat Jayaraman. Jive: Java interactive visualization environment. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOP-SLA '04*, pages 226–228, New York, NY, USA, 2004. ACM.
- [GJ05] Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization, Soft-Vis '05*, pages 95–104, New York, NY, USA, 2005. ACM.

- [HDD06] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *NODE 2006*, pages 17–32. GI, 2006.
- [JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, pages 273–282, New York, NY, USA, 2005. ACM.
- [JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 467–477, New York, NY, USA, 2002. ACM.
- [KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.
- [KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.
- [KFH13] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 352–361, Piscataway, NJ, USA, 2013. IEEE Press.
- [KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
- [KM04] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '04*, pages 151–158, New York, NY, USA, 2004. ACM.
- [KM08] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 301–310, New York, NY, USA, 2008. ACM.



- [KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006.
- [LCH<sup>+</sup>12] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 145–158, New York, NY, USA, 2012. ACM.
- [LD03] Bil Lewis and Mireille Ducasse. Using events to debug java programs backwards in time. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 96–97, New York, NY, USA, 2003. ACM.
- [Lew03] Bil Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, 2003.
- [LF95] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.
- [LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag.
- [LHS97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 304–317, New York, NY, USA, 1997. ACM.
- [LNZ<sup>+</sup>05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.
- [LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

- [MBP11a] Salman Mirghasemi, John J Barton, and Claude Petitpierre. Debugging by lastchange. Technical report, Technical Report. EPFL-REPORT-164250, 2011.
- [MBP11b] Salman Mirghasemi, John J. Barton, and Claude Petitpierre. Querypoint: Moving backwards on wrong values in the buggy execution. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, pages 436–439, New York, NY, USA, 2011. ACM.
- [MCE<sup>+</sup>02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.
- [Mir12] Salman Mirghasemi. *Querypoint Debugging (Semi-Automated Inspection of Buggy Execution)*. PhD thesis, IC, Lausanne, 2012.
- [Moh88] T. G. Moher. Provide: A process visualization and debugging environment. *IEEE Trans. Softw. Eng.*, 14(6):849–857, June 1988.
- [MS10] Del Myers and Margaret-Anne Storey. Using dynamic analysis to create trace-focused user interfaces for ides. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10*, pages 367–368, New York, NY, USA, 2010. ACM.
- [MT03] Kazutaka Maruyama and Minoru Terada. Debugging with reverse watchpoint. In *Proceedings of the Third International Conference on Quality Software, QSIC '03*, pages 116–, Washington, DC, USA, 2003. IEEE Computer Society.
- [ND86] Donald A. Norman and Stephen W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986.
- [NW94] Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94*, pages 313–325, New York, NY, USA, 1994. ACM.
- [PL88] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, PADD '88*, pages 124–129, New York, NY, USA, 1988. ACM.

- [PO11] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*, pages 199–209, New York, NY, USA, 2011. ACM.
- [PT09] G. Pothier and E. Tanter. Back to the future: Omniscient debugging. *Software, IEEE*, 26(6):78–85, Nov 2009.
- [PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07*, pages 535–552, New York, NY, USA, 2007. ACM.
- [RDB99] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.
- [RR05] Steven P. Reiss and Manos Renieris. Demonstration of jive and jove: Java as it happens. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pages 662–663, New York, NY, USA, 2005. ACM.
- [Sai05] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG'05*, pages 69–76, New York, NY, USA, 2005. ACM.
- [SF89] C. B. Stunkel and W. K. Fuchs. Trapedts: Producing traces for multicomputers via execution driven simulation. In *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '89*, pages 70–78, New York, NY, USA, 1989. ACM.
- [SPTH14] B. Siegmund, M. Perscheid, M. Taeumel, and R. Hirschfeld. Studying the advancement in debugging practice of professional software developers. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 269–274, Nov 2014.
- [SSA<sup>+</sup>12] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

- [Tas02] Gregory Tasse. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project, 7007(011)*, 2002.
- [TR81] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, September 1981.
- [ULF97] David Ungar, Henry Lieberman, and Christopher Fry. Debugging and the experience of immediacy. *Commun. ACM*, 40(4):38–43, April 1997.
- [Voa92] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Softw. Eng.*, 18(8):717–727, August 1992.
- [VZBJ04] Bradley T. Vander Zanden, David Baker, and Jing Jin. An explanation-based, visual debugger for one-way constraints. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology, UIST '04*, pages 207–216, New York, NY, USA, 2004. ACM.
- [WD09] W Eric Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, 9, 2009.
- [Wei79] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979. AAI8007856.
- [WR04] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *Proceedings of the 26th International Conference on Software Engineering, ICSE '04*, pages 512–521, Washington, DC, USA, 2004. IEEE Computer Society.
- [XQZ<sup>+</sup>05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.
- [XRTQ07] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07*, pages 85–94, New York, NY, USA, 2007. ACM.

- [Zel71] Marvin Zelkowitz. *Reversible Execution As a Diagnostic Tool*. PhD thesis, Cornell University, Ithaca, NY, USA, 1971. AAI7117676.
- [Zel73] M. V. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566–, September 1973.
- [Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.
- [Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [ZG04] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 94–106, New York, NY, USA, 2004. ACM.
- [ZG05] Xiangyu Zhang and Rajiv Gupta. Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.*, 2(3):301–334, September 2005.
- [ZGZ03] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.