6th International Conference on Internet Technology and Secured Transactions, 11-14 December 2011, Abu Dhabi, United Arab Emirates

Towards an Automatic Exploit Pipeline

Jared D. DeMott
Computer Science Dept.,
Michigan State University
East Lansing, MI 48895
jdemott@vdalabs.com

Richard J. Enbody
Computer Science Dept.,
Michigan State University
East Lansing, MI 48895
enbody@cse.msu.edu

William F. Punch
Computer Science Dept.,
Michigan State University
East Lansing, MI 48895
punch@msu.edu

***Abstract*—A continuous and fully automated software exploit discovery and development pipeline for real-world problems has not yet been achieved, but is desired by defenders and attackers alike. We have made significant steps toward that goal by combining and enhancing known bug hunting and analysis techniques. The first step is the implementation of an easy-to-use distributed fuzzer. Single fuzzers take too long to produce the number of results required. Since distributed fuzzers achieve high-output (typically many found bugs) sorting is required, which we include. We add another layer of triage support by combining in an enhanced fault localization process. Our work automates much of the process so that human resources are only needed at a few key checkpoints along the pipeline, arguably enhancing overall system efficiency. We demonstrate our process on contrived code, the Siemens suite, and two real-world pieces of code: Firefox and Java.**

Keywords- Software Testing and Debugging, Software Security, Distributed Fuzzing, Fault localization, and Automatic Vulnerability Discovery and Exploitation

I. INTRODUCTION

Software exploit development is a hard problem, and current state of the art approaches rely on the creativity of the people that do the work. For example, a security researcher has a hunch about a potentially buggy feature in a product and decides to investigate. Reverse engineering, fuzzing, and whatever other tools they find useful are applied to find a bug. Once a raw bug has been discovered, the quality of the bug must be determined. In this context a raw bug is being referred to as a crash found via fuzzing, but not yet analyzed to determine the source or severity of the bug. If the bug appears exploitable, the exploit developer begins the arduous task of creating a stable attack. Last, but certainly not least, the researcher must also bypass whatever system protections are in place for the attack to actually work.

Given the laborious nature of this work, it is important to automate the routine aspects of the process so as to give the creative human the freedom to focus on certain key portions of the process. Prior research [10] has shown methods to automatically create an exploit given certain types of crashes. Researchers at Carnegie Mellon [3] furthered this work and used static analysis techniques to discover bugs before attempting to automate exploitation. Our goal complements both prior works by automating early stages of the pipeline and by developing techniques that work on real-world problems. This is a particularly important point since current exploitation research has not been sufficiently powerful to operate on arbitrary real-world problems. Thus, the primary objective of this study is to automatically find high quality vulnerabilities in real-world code so the analyst is free to focus on the exploit, rather than laboriously sifting through multitudes of irrelevant crash data looking for the rare, but insightful, element(s). High quality bugs are those that are reliable (trigger each time) and severe. Severe indicates a reasonable chance that the bug can be exploited by attackers.

We believe that reducing the triage time of discovered bugs could save more money (time) than partial exploit automation. In fuzzing-based systems, more hours are often spent triaging then in exploit development. Thus, the further upstream one can automate the process, the greater the impact this automation will have on the overall system. Existing automatic exploit development work [3, 10] focuses too much on trivial problems on systems without generic exploit mitigations. More general approaches are required to reduce human interaction while at the same time increasing productivity

II. CONTRIBUTIONS

In this article we present an automated software bug discovery and analysis framework ("exploit pipeline"), which combines:

- A distributed fuzzer that is powerful enough to create numerous bugs from real-world applications (even zero-day vulnerabilities).
- Prioritizing of fuzzer output (bugs) based on severity.
- An integrated and improved (noise tolerant) control-flow fault localization to decrease triage time.
- A GUI that allows junior analysts to steer the process, while allowing senior researchers to work at a few critical points.

III. BACKGROUND

Bugs are discovered, reported, analyzed, repaired, and exploited in many ways. In this section, we focus on the techniques used to find exploitable vulnerabilities. There are a number of techniques employed by present day security researchers for bug discovery. They include: symbolic analysis, automatic exploit development, fuzzing, crash analysis, and fault localization.

A. Symbolic Execution

Symbolic execution refers to the analysis of programs by tracking symbolic rather than actual values, an example of abstract interpretation. Using formal verification techniques as in [3] to find bugs is an effective approach. However, source code is usually required, and large code bases are often infeasible to analyze. In [3], researchers were able to upgrade their tool set to handle medium sized programs by focusing on potentially exploitable paths, but they still required source code. In this context medium sized programs would typically be command line programs with thousands of lines of code. Large programs would be millions of lines, such as web browsers and other heavy weight desktop applications.

B. Automatic Exploit Development

Automatic exploit development research is a relatively new field of study, but one that is very important. The object is to begin with a valid bug, typically a memory corruption crash, and generate a working control flow hijack [3, 10]. In [3], researchers showed how both stack overflows and format bugs could be handled, and in [10] researchers showed how stack overflows, function pointer overwrites, and wild writes could be handled. As a comparison, [10] required a bug to begin with while [3] attempted to find bugs via static analysis. Neither can bypass common generic operating system protections such as non-executable data memory and address space randomization. Additionally, large programs pose a problem to both approaches since they rely on constraint solvers and symbolic execution. Fuzzing is often an attractive alternative, since it can handle real-world problems.

C. Fuzzing

Fuzzing is another potential stream of raw bug discovery, and is the one we chose to base the pipeline we describe in this paper. Fuzzing is a testing technique known for its ability to uncover memory corruption bugs which tend to have security implications [21]. Hackers and security researchers have used fuzzing for years to finds bugs in widely available commercial and open source software. Because of its effectiveness, fuzzing is included in many development environments, e.g. Microsoft incorporates fuzzing into their Secure Development Lifecycle [15].

D. Crash Analysis

Crash analysis is a security-oriented branch of debugging. Once a bug discovery tool (such as a fuzzer) has identified a bug, the relative merit (quality) of the resulting crash needs to be determined. This effort has traditionally been a manual task, involving a debugger used alongside a review of the pertinent code regions. Recently, Miller et al. [16] released a set of tools that build on a binary analysis system called BitBlaze [5] with the intent to speed up manual crash analysis in a way similar to existing debugger plugins. They did not fully automate the process, but they did aid the human by removing mental effort for complex debugging tasks.

Another crash analysis tool, and the one we employ in our framework, is a debugger plugin called !exploitable [7]. The !exploitable tool is a Windows debugging (Windbg) extension that provides automated crash analysis, security risk assessment and guidance. The tool uses a hash to uniquely identify a crash and then assigns one of the following exploitability ratings to the crash: Exploitable, Probably Exploitable, Probably Not Exploitable, or Unknown.

E. Fault Localization

Fault localization (FL) attempts to decrease the time spent debugging by automatically directing testers to the most likely locations of the underlying reason(s) for the fault. Here we review an approach which is popular and on which we build.

1) Tarantula

Researchers at Georgia Tech have used portions of runtime information to develop a technique they call Tarantula [14, 12, 19]. Tarantula uses *code coverage* information to track each line of code as it is executed. A pass-fail oracle labels each line as to whether the line participated in a fault or not. The intuition behind Tarantula is that entities (code) in a program that are primarily executed by failed test cases are more likely to be at fault than those that are primarily executed by passed test cases. However, unlike prior studies [1], Tarantula allows some tolerance for lines that occasionally participated in passed test cases. This tolerance has been shown to make the technique very effective.

Table I. Sample Coverage Data (1=covered, 0=not covered)

	Run 1	***Run 2***	Run 3	***Run 4***	Suspiciousness
Line 1	1	0	1	1	.58
Line 2	0	1	1	1	***.82***
Line 3	1	0	1	0	0
Line 4	0	1	1	0	.58
Line 5	1	0	0	1	.58

Table 1 provides sample code coverage data to be evaluated by Tarantula using the adapted Ochiai statistical formula which is described by (1). Runs 2 and 4 result in faults. Run 3 has no faults, but executes the same program statements that a faulty run executed (lines 2 and 4). The "probability" that any one line is the fault source is calculated as follows:

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{totfailed\ x\ (failed(s) + passed(s))}} \qquad (1)$$

In (1), *failed(s)* is the number of failed test cases that executed statement *s* one or more times. *Passed(s)* is the number of passed test cases that executed statement *s* one or more times. *Totfailed* is the total number of failed test cases for the entire test suite. If the denominator is zero, zero is assigned as the result. Using this calculation, Tarantula identifies Line 2 as the likely cause of the fault.

Using the suspiciousness score, the covered lines are sorted. The set of lines with the highest score are considered first by the programmer attempting to fix the bug. If after examining the first entity the bug is not found, the programmer continues down the list from highest to lowest score. Ties must be

manually resolved. For a visual cue Tarantula also colors code for further debugging help.

Path-based approaches such as these are very effective with low overhead. However, they have three common weaknesses. First, while they work well for control-flow bugs, they fail to find data-only bugs where there is no path difference between good and bad traces. Second, path-based algorithms may become confused in the presence of noise since unique code blocks which are not relevant, but happen to show up in bad traces, could be incorrectly selected as the actual fault. Third, many prior FL implementations require source code and a test suite of good and bad runs, though there has been work to automatically create tests if they are lacking [2]

IV. Distributed Fuzzing

Distributed fuzzing has been shown to be effective by [9] and [17]. We have designed a pipeline approach called ClusterFuzz (CF), which includes a distributed fuzzing system. The CF pipeline works as follows:

- First, the system automatically generates crashes via fuzzing.
- Second, the system sorts the crashes so that the high quality crashes can be serviced first. In this context, high quality indicates a crash that is both reliable (repeatable) and severe. Severe in this context indicates that the crash appears to be exploitable. Crashed process state is used to make the severity suggestion. The severity rating is not completely accurate, but good enough for a first round of sorting.
- Third, on the best crashes the system attempts to provide the actual cause of the crash using a statistical, path-based FL algorithm.

A. ClusterFuzz

CF is a framework that facilitates fuzzing in an effort to discover previously unknown software crashes. Our GUI enables workers of all skill levels to effectively look for bugs. Importantly, CF does fuzzing *in parallel*, making fuzzing more efficient and taking advantage of modern hardware. CF *does not* require either source code or a test set to generate tests. Once an application has been fuzzed, and crash results have been generated, CF proceeds to do further evaluation of the results, in particular rating, ranking, and analyzing the crashes. Fig. 1 shows the high level design for CF. Each of the major sections from the diagram is briefly described.

Configure. The first thing the bug hunter does is choose a target to fuzz. That target, a.k.a. the subject under test (SUT), must then be properly installed in a "base" virtual machine (VM). We use Windows XP for the base VM operating system (OS) as it is easier to configure and requires fewer resources then Windows Vista or 7. Windows in general was chosen both because the Peach [18] tool runs best under Windows, and because many interesting targets to fuzz, such as Microsoft's Internet Explorer, reside only on Windows[1]. Next, the data model (the *Peach Pit*) must be constructed specifically for the SUT. Our GUI includes an option to automatically create a mutation Peach Pit (fuzzer configuration). While the base pit often yields reasonable results, and provides a starting point for deeper investigation, it is also typical to run the base mutation pit while simultaneously spending time constructing a more complete data model.

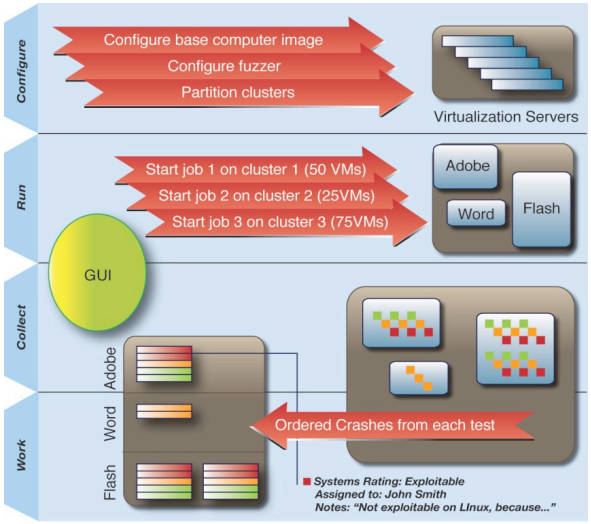

Figure 1. ClusterFuzz

Run. Our present setup has six VMware ESX 4.0 virtualization servers. Based on our current hardware, each of our servers can handle approximately 25 virtual machines cloned from the base image. Therefore, we have 150 VMs at our disposal (we spent $30,000USD on hardware). Each virtual machine has generous amounts of RAM and CPU cycles, due to the continued falling cost of hardware. These VMs may be used for a variety of purposes such as:

- Auto-MinSet, which uses code coverage to find an optimal test set from samples automatically downloaded from the internet
- Auto-Retest, which checks the reliability of crashes
- Execution Mining (described later)
- Fuzzing

Collect and Report. Once the runs have completed the ordered crash results are collected. The CF GUI provides a convenient way for testers or exploit developers to view each result and log the progress of the overall process.

B. Results

The following are applications and data formats that have been fuzzed using ClusterFuzz:

- Client-side applications
  - Browsers
    - Internet Explorer, Chrome, Firefox, Safari, Opera
  - Office Applications
    - Writer (Open Office), Word (Microsoft Office), Adobe Reader, Adobe Flash Player, Picture Manager (Microsoft Office)
  - Other
    - iTunes, QuickTime, Java, VLC Media Player, Windows Media Player, RealPlayer

[1] It is possible to use a different base OS and a different fuzzer with ClusterFuzz.

- File formats:
  - Images:
    - JPG, BMP, PNG, TIFF, GIF
  - Video:
    - AVI, MOV
  - Office:
    - DOC, DOCX, XLS, XLSX, ODT
  - Adobe:
    - PDF, SWF

We gathered statistics over one month of running CF on the previously listed applications, providing many of the file formats as input. Not every combination produced a fault, but when faults were noted we recorded the collective results, which are as follows:

- 141,780 crashes total
  - Crashes/day: 4726
  - Crashes/hr: 197
- 828 total unique crash bins
  - 17 "Exploitable" bins
  - 6 "Probably exploitable" bins
  - 0 "Probably Not exploitable" bins
  - 805 "Unknown" bins
  - Unique crash bins/day: 28
  - Unique bins "probably exploitable" or "exploitable" /day: 0.**9**

The ratings shown above are those reported by !exploitable. The data shown helps illustrate how !exploitable works, and the importance of filtering in fuzzing. While there may be many crashes, often there are many fewer unique *and* security-critical bugs. These high quality bugs are what developers and bug exploiters usually want to focus on. In this case, we define high quality (HQ) as: Reliable and Severe (probably exploitable or higher).

On top of the bucketing provided by !exploitable, we also collect and store relevant crash information such as the registers used and nearby disassembly code in the database. This allows researchers to later search for particular register patterns as they become widely known. For example, if the Intel registers ECX and EIP contain the same value that condition may indicate the presence of an exploitable C++ structured exception handler (SEH) overwrite.

Even with a rough sorting tool and better storage of crash information, reversing one crash from each bin and further sorting the unknowns is a substantial amount of manual work (step 4 from Fig. 1). We continue the automated sieving process in the pipeline by automatically detecting the underlying root flaw in each case. Our first step toward that goal is achieved by including fault localization as described in the next section

V. Integrated and Enhanced Fault Localization

In section A we discuss the implementation of our fault localization (FL) system, which is a part of ClusterFuzz. We call our FL system Execution Mining (EM). In section B we describe novel enhancements to a path-based approach using noise-canceling heuristics. Section C details the results of our experiments.

A. *Automatic Fault Localization and Visualization*

Fuzzing identifies the existence of faults, but does not identify what error was the underlying cause of the fault, or help the reverse engineer to do so. EM is our bug-focused test set creation, fault localization, and visualization process. Fig. 2 shows an overview of our fault localization technique.

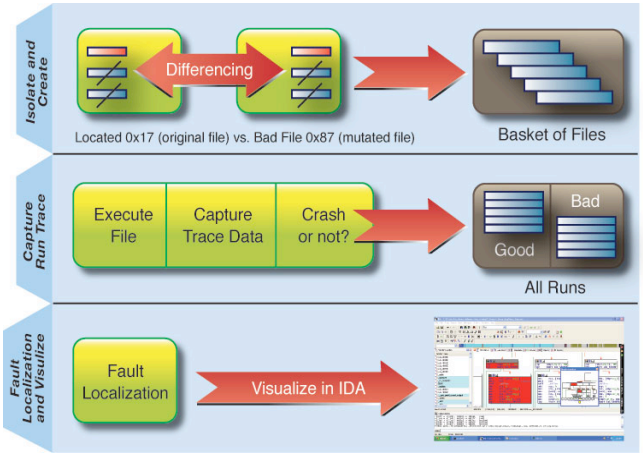

Figure 2. Execution Mining

The first step is to create a set of good and bad inputs (files)—necessary for statistical fault localization. Following the process shown in Fig. 2, we first isolate the mutated byte(s) in the bad input file that caused the application to crash. Note that CF typically is employed on binary files for client-side applications. The tool changes each difference in the bad input file back to its original value, until the bad input file no longer crashes the application. Such a location is assumed to hold the offending data. If, for example, the offending data is a single byte, 256 new files can now be created using all possibilities of that byte. The file-and-byte assumptions are reasonable for parallel client-side fuzzing—our current focus.

Now that we have a set of good and bad files, each file can be run through the trace collection software. A basic block (BB) trace collection is created and tagged as either crashed (bad) or not (good). After all of the traces have been collected, the fault localization is run as shown at the bottom of Fig. 2. The top fault locations are then colored and visualized in IDA pro [11], a popular reverse engineering tool. The IDA pro disassembly screen is centered (redirected) on the most suspicious fault location.

We found that for large programs the fault localization was not always as effective as we would like. The primary reason for this was found to be the existence of "noise" that deflects the ability of path-based approaches to discover the important flawed blocks. Noise indicates too many chance code blocks driving the FL algorithm towards false conclusions. In the next sections we describe our enhancement and the results.

B. *Execution Mining Algorithm Details*

Algorithm 1 details the high level process from a discovered bug, until the final ordered list of suspicious basic block

locations is output. Afterwards, the output file is loaded into IDA pro for visualization.

Algorithm 1. Statistical Fault Localization with Noise Canceling

Input: Data Sample (*Original*). A fault is found by changing the sample to create a mutated sample (*Mutated*).
Output: Ordered list of suspicious basic block addresses and supporting data (*OrderedSuspiciousList*).

```
good = {} #BasicBlockAddress: RunsCount, TimesExecuted
bad = {} #BasicBlockAddress: RunsCount, TimesExecuted

Subroutine FilterBasicNoise (OrderedSuspiciousList):
  for each basic block address k and score s in
OrderedSuspiciousList do
      good_runs_count, good_times_executed = good[k]
      bad_runs_count, bad_times_executed = bad[k]
      if (good_runs_count + bad_runs_count) < (.1 * total_blocks)
      then
            s -= medium_modifier
      end
      if (good_times_executed != bad_times_executed)
      then
            s += small_modifier
      end
      OrderedSuspiciousList_FL-Enhanced[k] = s
  end
Return OrderedSuspiciousList_FL-Enhanced

Main:
  FileOffset = LocateDifference (Original, Mutated);
  TestSet = CreateSimilarDataInputs (Original, Mutated,
FileOffset);

  for each input file f in TestSet do
     RetVal= CreateTraceFile (f, Pintool, TraceName, TraceDir, ...);
     if (RetVal) == CRASH
     then
           AppendLabelsFile_Crashed (TraceName);
     else
           AppendLabelsFile_NoCrash (TraceName);
     end
  end

  for each trace t in TraceDir do
     if (t) == CRASH
     then
           UpdateDictionary(bad, t);
     else
           UpdateDictionary(good, t);
     end
  end

  OrderedSuspiciousList_FL-Basic = OchiaiBased_FL (bad, good);

  OrderedSuspiciousList_FL-Enhanced = FilterBasicNoise
(OrderedSuspiciousList_FL-Basic);
```

Note that each of the *OrderedSuspiciousList* arrays is output as a file. We call them FL-Basic and FL-Enhanced. FL-Basic is the base fault localization output. FL-Enhanced adds enhancements to filter out noise associated with large amounts of basic-block tracing. This filtering is achieved by adding score modifiers. The most effective modifier adds a penalty for basic blocks that do not appear in a statistically significant amount of runs (i.e. we presume that the block was noise). The other modifier shown in the *FilterBasicNoise* subroutine of Algorithm 1, increases a basic block's suspiciousness score by a smaller amount if the number of times that block was executed differs between good and bad traces. These techniques may seem simple, but are shown effective when combined.

C. Results

We experimentally show that our new approach is valid and operates better than a basic statistical fault localization approach would alone. Our enhanced approach matches that of another FL algorithm, but does so as part of a larger real-world focused analysis system, and does so without the need to mine graphs as in [4]. We show results on contrived problems, and on the popular Siemens test set similar to [4]. We also test against large, real-world problems.

1) Contrived

In our initial research we wanted to show that our theory was valid. Thus, we operated our fault localization process shown in Algorithm 1 against various small and contrived problems and were able to determine the root cause more effectively when using the FL-Enhanced output.

2) Siemens Benchmark

We tested our FL approaches using the Siemens test set [20], which is commonly used to determine the effectiveness of fault localization techniques. The Siemens test set originates from work done at Siemens Corporate Research on dataflow and control flow test adequacy [13]. Table 2 shows that FL-Enhanced is able to match the effectiveness of one of the best fault localization techniques, Top-K LEAP [4], while FL-Basic underperformed.

Table II. Comparing FL Types (1=success; 0=failure)

	FL-Basic	FL-Enhanced	Top-K LEAP
print_tokens2: v10	1	1	1
replace: v23	1	1	1
schedule: v1	0	1	1
schedule: v5	0	1	1
schedule: v6	0	1	1
schedule2: v8	0	0	0
tot_info: v18	1	1	1
tot_info: v20	0	1	1
Average	37.50%	87.50%	87.50%

Note that it was not possible to use all the versions of each application provided in the Siemens test set, since our system is only looking for applications that exhibit memory corruption bugs (typically an access violation). This subset of all bugs is a critical set to those hunting for exploitable bugs, such as the work done by penetration testers. Second, note that the schedule2, version 8 bug is not properly localized by any of the approaches. By not localized we mean, not in the top 5 suspicious locations. (We assume the tester will tire, if too many results must be examined.) We spend the rest of this section examining the difficult bug in schdeule2:v8.

The particular bug in question is a missing-code issue. The routine within the schedule2 program with the error is a function called *put_end*. The code is shown in Code Listing 1.

CODE LISTING 1. put_end function

```
1.  int put_end(int prio, struct process * process) /* Put process at
    end of queue */
    {
2.    struct process **next;
      /*if(prio > MAXPRIO || prio < 0) return(BADPRIO); */
       /* find end of queue */
3.  for(next = &prio_queue[prio].head; *next; next = &(*next)->next);
4.    *next = process;
5.    prio_queue[prio].length++;
6.    return(OK);
    }
```

Between lines 2 and 3 we see the error, a commented input validation check that needs to be uncommented. FL-Basic found the erroneous area as the 19th choice and FL-Enhanced as the 12th pick, showing a 37% improvement in rank.

3) Firefox

To test our localization outputs against a very large, real-world program the open source Mozilla Firefox browser [8] was chosen. We used version 3.5.8 because there was a public PNG (graphic format) bug, which had a detailed description on Bugzilla detailing the manual process developers followed to determine the root of the crash [6]. The cause of the bug comes from code in the file nsPNGDecoder.cpp in the location ../modules/libpr0n/decoders/png/. The *row_callback* function (line 684, revision 1.9.1) omits an important check to see if the *row_num* variable is in bounds. Code Listing 2 shows the relevant snippet from the repaired revision 1.9.2, with the added code on lines 5 and 6.

CODE LISTING 2. row_callback function

```
1.  void row_callback(png_structp png_ptr, png_bytep new_row,
    png_uint_32 row_num, int pass)
    {
2.    nsPNGDecoder *decoder =
    static_cast<nsPNGDecoder*>(png_get_progressive_ptr(png_ptr));
      // skip this frame
3.    if (decoder->mFrameIsHidden)
4.      return;
5.    if (row_num >= decoder->mFrameRect.height)
6.      return;
    }
```

The primary reason this bug is difficult to localize with path-based approaches alone is that in large programs there is simply too much noise and not enough signal. By noise we mean seemingly important, but actually chance occurrences, of blocks executed in bad traces, which happen to not be executed in good traces. By signal we mean the actual unique paths in bad traces, which we are looking for. The task of fault localization presumes there is a reasonable signal to noise ratio. Conversely, if code is widely variable between runs, even with similar inputs, the Ochiai algorithm may key in on unrelated, but unique code from bad traces, as the source of the bug.

Therefore, it is necessary to hone in on the portion of large code bases that is interesting thereby reducing the variability in analysis. One technique that we employ is to note which module (DLL in Windows) the exception occurs in, and only perform the FL within that module. That solves many variability issues. However, in this case Firefox uses a very large module called *libxul*, which is a combination of many other libraries designed to decrease the start time when Firefox is launched, making the code to analyze still very large. Our FL-Basic technique found the correct function as the 131st choice in the ordered list of suspicious locations. FL-Enhanced found it as the 66th choice, a 50% improvement.

4) Java

To show that our system finds new real-world bugs (sometimes called zero-day vulnerabilities), multiple previously undisclosed flaws were identified in the latest java version (1.6.0_25) when fuzzing the Java picture parsing routines by using sample JPEG pictures. The flaws reside in the color management module (CMM.dll) of the Windows Java package, specifically in the *SpTagToPublic* function. Code Listing 3 shows the area of code where one of the bugs resides. Lines 11-16 contain the bug. *Buf* is incremented for certain character inputs. Later in lines 18-25 *Buf* is used in a memory copy, and data outside the bounds of *Buf* may be inserted into the destination buffer. It is possible to use this type of vulnerability to leak internal program information. Leaking the location of a module is a recent approach used by attackers as part of a multi-part technique to bypass modern protections (non-executable data memory and address space randomization).

CODE LISTING 3. SpTagToPublic function

```
1.  #define IS_VALID_SIGNATURE_CHAR(c) ((0x30 <= ((unsigned
    char)(c)) && ((unsigned char)(c)) <= 0x39) ||  (0x41 <= ((unsigned
    char)(c)) && ((unsigned char)(c)) <= 0x5a) ||  (0x61 <= ((unsigned
    char)(c)) && ((unsigned char)(c)) <= 0x7a))
    /*DESCRIPTION  -- Convert attribute value from more compact
    internal form. */
2.  SpStatus_t KSPAPI SpTagToPublic ( SpTagId_t TagId, KpUInt32_t
    TagDataSize,  SpTagValue_t FAR *Value, …)
    {
3.        char        KPHUGE *Buf;
4.        char        KPHUGE *BufSave;
5.        KpUInt32_t      Index, Limit;
6.        SpData_t        FAR *Data;
7.        SpSig_t         TypeSig;
8.        KpUInt32_t            trimCount = 0;
    /* set tag id and type in callers structure */
9.        Value->TagId = TagId;
    ...
10.       Buf = TagData;
    /* trim tag signature if needed */
11.     while (!IS_VALID_SIGNATURE_CHAR(*Buf)) {
12.       if (++trimCount > TagDataSize) {
13.         return SpStatBadTagData;
14.       }
15.       Buf ++;
16.     }
    ...
```

```
17.   switch (Value->TagType) {
      ...
18.     case Sp_AT_Unknown:
19.           Value->Data.Binary.Size = TagDataSize;
20.           BufSave = (char KPHUGE *) SpMalloc (TagDataSize);
21.           if (NULL == BufSave)
22.                 return SpStatMemory;

23.           KpMemCpy (BufSave, (void *)(Buf - 8), TagDataSize);
24.           Value->Data.Binary.Values = BufSave;
25.           return SpStatSuccess;
      ...
```

In this case, FL-Basic and FL-Enhanced both correctly identify the bug within the first five choices. The coloring and fault localization are helpful since the function where the bug exists is complex. Fig. 3 shows the visualization and bug localization for Code Listing 3.

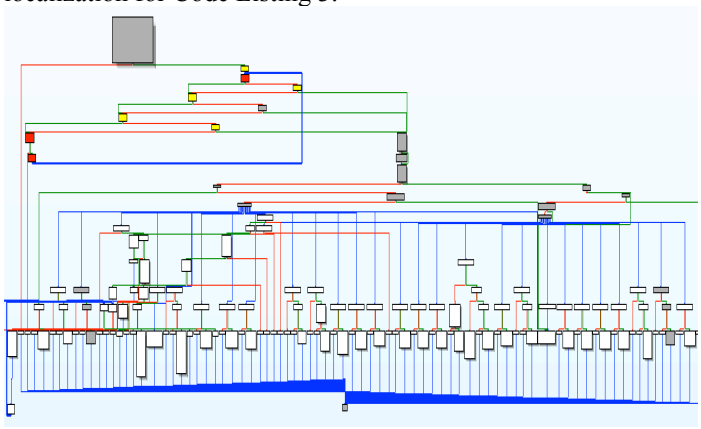

Figure 3. Visualization of Java bug (*SpTagToPublic function)*

The small red and yellow blocks (upper left of Fig. 3) indicate the most suspicious blocks (lines 11-16 from Code Listing 3). The grey blocks indicate basic blocks executed by good and bad traces. The white blocks were not executed. These colorations help should further manual analysis of difficult bugs be required.

VI. Conclusions

We believe that understanding the capability of attackers is paramount in forming a proper defense of software. We believe that we, and others before us, have shown that it is possible to create an exploit pipeline. As greater portions of that pipeline are automated, the threat of actionable zero-day exploits increase. Conversely we believe our pipeline can also be used to inform development teams on the amount, severity, and location of critical bugs.

Our easy-to-use distributed fuzzing system shows that mass bug hunting is practical for a modest cost. Crash analysis and fault localization of bugs decrease the amount of time analysts need to spend in the upstream triage phase. We believe we have shown a probable roadway toward a fully automated exploit pipeline, particularly as future work continues on automating real-world exploit development.

We have also shown our enhanced fault localization is an improvement over basic Tarantula, and demonstrated that it aids in crash analysis as part of a larger bug hunting system. For future work, we plan to use data flow tracking to continue to counter noise in an effort to improve FL results.

References

[1] AGRAWAL, H., HORGAN, J.R., LONDON, S., WONG, W.E. 1995. Fault localization using execution slices and dataflow tests, *Software Reliability Engineering. Proceedings., Sixth International Symposium on*, vol., no., pp.143-151

[2] ARTZI, S., DOLBY, J., TIP, F., and PISTOIA, M. 2010. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis* (ISSTA '10). ACM, New York, NY, USA, 49-60.

[3] AVGERINOS, T., CHA, S., K., HAO, B., L., T., and BRUMLEY, D. 2011. AEG: Automatic Exploit Generation. *18th Annual Network and Distributed System Security Symposium*. San Diego, CA.

[4] CHENG, H., LO, D., ZHOU, Y., WANG, X., and YAN, X. 2009. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (ISSTA '09). ACM, New York, NY, USA, 141-152

[5] BITBLAZE. 2011. http://bitblaze.cs.berkeley.edu/

[6] BUGZILLA. 2011. https://bugzilla.mozilla.org/show_bug.cgi?id=570451

[7] EXPLOITABLE. 2011. http://msecdbg.codeplex.com/

[8] FIREFOX. 2011. http://www.mozilla.org/projects/firefox/

[9] GODEFROID, P., AND MOLNAR, D. 2010. Fuzzing in the Cloud (Position Statement). http://research.microsoft.com/apps/pubs/default.aspx?id=121494

[10] HEELAN, S. 2009. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. *MSc Computer Science Dissertation, Oxford University*.

[11] HEX-RAYS. 2011. http://www.hex-rays.com/idapro/

[12] HSU, H., JONES, J. A., and ORSO, A. 2008. Rapid: Identifying Bug Signatures to Support Debugging Activities. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (ASE '08). IEEE Computer Society, Washington, DC, USA, 439-442.

[13] HUTCHINS, M., FOSTER, H., GORADIA, T., and OSTRAND, T. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering* (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 191-200.

[14] JONES, J., A., and HARROLD, M., J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (ASE '05). ACM, New York, NY, USA, 273-282.

[15] MICROSOFT. 2011. http://www.microsoft.com/security/sdl/

[16] MILLER, C., CABALLERO, J., JOHNSON, N., M., KANG, M., G., MCCAMANT, S., POOSANKAM, P., SONG, D. 2010. Crash Analysis with BitBlaze. *Black Hat USA*

[17] NAGY, B. 2009. Finding Microsoft Vulnerabilities by Fuzzing Binary Files with Ruby - A New Fuzzing Framework. http://www.coseinc.com/en/index.php?rt=download&act=publication&file=A%20New%20Fuzzing%20Framework.pptx

[18] PEACH. 2011. http://peachfuzzer.com/

[19] SANTELICES, R., JONES, J., A., YU, Y., and HARROLD, M., J. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 56-66.

[20] SOFTWARE-ARTIFACT INFASTRUCTURE AND REPOSITORY (SIR). 2011. http://sir.unl.edu/portal/index.html

[21] TAKANEN, A., DEMOTT, J., MILLER, C. 2008. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc. Norwood, MA.