Memory Safety Bug Hunting with BAP

Tiffany Bao, Dominic Chen, Rijnard van Tonder

April 29, 2015

1 Introduction

The Problem

Although numerous debugging tools have been developed for performing static and dynamic software analysis, the majority of these require access to source code or utilization of special compiletime flags to embed debug information within the output binary. In contrast, little effort has been spent on developing static binary analysis tools which can directly analyze offtheshelf executables without executing them. For this project, we plan to focus on a subset of common programming bugs that tend to introduce security vulnerabilities, known as memorysafety errors. Examples of these include buffer overflows, double frees, and uninitialized variables. In particular, we define *Security Outcomes* with respect to certain memory-safety errors, and subsequently implement and test static analyses that yield these outcomes. All analyses were written against our group's Binary Analysis Platform (BAP) [2, 3].

A factor of this research area that appeals to many is the ability to conclude whether vulnerabilities do or do not exist by performing an analysis purely on the binary level. Due to the inherent difficulties of operating at this level, our approach to the problem starts out coarse, and continues to be incrementally refined as we become more familiar with the numerous barriers that inhibit vulnerability detection in binaries. In this respect, we have much to say towards lessons learned in §5.

1.1 Approach Overview

At the outset of this project, we planned to implement and test the static analyses with respect to our original proposal:

1. Intraprocedural data dependency resolution of instructions
2. Intraprocedural detection of unchecked memory allocations
3. Basic intraprocedural detection of buffer overflows in the stack
4. Basic intraprocedural detection of buffer overflows in the heap

Equipped with the knowledge of data flow dependency between instructions (1), we determined to reason about various properties of typical vulnerabilities that we could check (2-4). For example, we might like to follow how arguments to a 'strcpy' call are influenced by previous instructions as a function of the stack register.

1.2 Related Work

Engler et. al. found numerous bugs with system-specific static analyses, implemented by way of a meta-compilation language for the `gcc` compiler [4, 5]. Analyses are composed in the form of correctness rules and automatic inference of rules form source code. Memory errors that are checked include null pointer bugs, not checking allocation results, and uses of freed pointers. This relates to our goal in that we would ideally like to perform similar checks, albeit at the level of complexity of binary code. Whereas Engler's work matured into a generic platform for specifying analyses, we must start first with an ad-hoc approach to gain insight and familiarity in the problem domain presented by binaries.

1.3 Contributions

Our key technical contributions are as follows:

1. A dataflow analysis framework for BAP.
2. General plugins for BAP that deliver
   (a) reaching definitions and
   (b) data flow dependency resolution (use-def and def-use chains).
   (c) registers corresponding to function call arguments and returns, according to the ARM ABI.
3. Security-specific plugins for BAP that produce security outcomes by checking
   (a) whether memory allocation checks are performed on 'memcpy',
   (b) whether calls such as strcpy are safe from buffer overflows under specific conditions, and
   (c) whether malloc calls...
4. Results of testing the aforementioned plugins on the `GNU Coreutils` software suite.

1.4 Structure

Due to the exploratory nature of this project on the binary level, we concentrate much of our efforts on looking at viable approaches of analysis, rather than optimizing properties of the analysis or testing against measurable benchmarks. For this reason, it doesn't come as a surprise that our evaluation section may be less expressive than our approach and "lessons learned".

2 Approach

2.1 Overview

Here we discuss our approach toward ensuring security properties as it evolved over the course of the project. Initially, we determined to reason about each of the cases in §1.1 that relate to vulnerabilities (2-4). To do so, we needed some basic infrastructure (1) which we describe first. Thereafter, we describe our specific approaches to performing security analyses with respect to specific *Security Outcomes*.

2.2 The Case for Reaching Definitions

For example, consider the following code, where we desire to verify that the return value of `malloc` is checked:

Listing 1: malloc.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5    // Uninitialized variable
6    char *array = 0;
7
8
9    array = malloc(32 * sizeof(*array));
10
11   if (!array)
12     return -1;
13
14   free(array);
15
16   return 0;
17 }
```

The disassembly of this binary might appear as follows (with two basic blocks, before and after the call):

Listing 2: Malloc disassembly

```
begin(main_ENTRY)
    000082c8: 04 e0 2d e5    str lr, [sp, #-4]!
    000082cc: 20 00 a0 e3    mov r0, #0x20
    000082d0: 49 00 00 eb    bl #0x124          ; call malloc
end(main_ENTRY)

begin(main_0xc)
    000082d4: 00 00 50 e3    cmp r0, #0x0       ; check malloc return value
    000082d8: 02 00 00 0a    beq #0x8
end(main_0xc)
```

We would like to confirm that a check takes place after the call to malloc; i.e., we would check for the presence of the `cmp` instruction and the operand `r0` in `main_0xc`.

However, it may easily be the case that the contents of `r0` are first moved to another register, `r3`, which is subsequently checked by a `cmp` instruction. In order to verify the buggy condition that the malloc return value is never checked, we must verify that any flow of this data from `r0` to other registers are also not checked. To obtain the statements where `r0` data flows to, we make use of def-use chains. For the definition of `r0` in the BIL IR, we check all uses of `r0` and transitively look up the def-use chains for further registers, such as `r3`, which the value in `r0` flows to. On the final set of instructions, we verify that the data is subject to a `cmp` instruction.

This example demonstrates the need for def-use chains; our stack and heap based analyses in turn also require use-def chains. Due to retreating edges in the CFG, it is difficult to produce def-use chains given a single instruction. For this reason we first implemented a dataflow framework and a reaching-definitions pass. From the reaching definitions, we generate def-use and use-def chains as necessary—e.g., we filter on those reaching definitions which concern a particular use or def of a register.

2.3 Data Flow Dependency

An extension to the def-use and use-def chains was to produce the data (flow) dependencies of a given statement in a forward or backward direction, respectively. Essentially, given a register in a definition (such as a Move instruction in the BIL IR), we can produce a set of all instructions that are determined to transitively affect its value. We implement this by way of a data dependency plugin for BAP. Consider the arguments to the following `strcpy` call:

Listing 3: strcpy.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv) {
  char dst[10] = {0};
  strcpy(dst, argv[1]);
}
```

For the `dst` variable on the stack, encoded as an `<address,statement>` pair, we have:

```
<(0x8490:32, 0)> R0 := R2
```

The plugin will determine all preceding instructions it is dependent on:

```
<(0x8444:32, 3)> SP := SP - 0x8:32
<(0x8448:32, 2)> R11 := SP + 0x4:32
<(0x848C:32, 2)> R2 := R11 - 0x10:32
```

Now, we can observe the role that the stack pointer `SP` plays when considering `dst`. A full listing of the BIL disassembly, the data dependencies for `argv[1]`, and a graphical output may be viewed in the Appendix A.

2.4 Malloc Checking Security Outcomes

2.5 Stack-based Security Outcomes

A number of unfruitful approaches documented in §5 lead us to opting for an approach where we identify two properties which are important from a security perspective, without inferring sizes of stack-based buffers.

`src` is a constant If we can determine that the `src` buffer is a constant, (and therefore not dependent on any user input), the `strcpy` call is safe. While a good property to check, it was found that a constant value corresponding to the `src` argument didn't occur in practice. Depending on compiler optimizations `strcpy` (as well as `sprintf` calls) would be compiled to a `LDMIA.W` ARM instruction which directly copies memory. Lastly, we observed that `strcpy` and `sprintf` may be replaced by the `__strcpy_chk` and `__sprintf_chk` counterparts, which are less interesting to analyse from a security standpoint.

Nevertheless, we implemented a partial check of this case in the plugin, and assumed the standpoint that any `src` address originating from the `ro.data` section implies a constant, fixed value. However, for this check we only consider the statement which loads a value immediately before the call, and not originating previous statements.

Stack accesses within the frame For this outcome, we confirm that all instructions which access elements on the stack, plus a fixed offset, are within the stack range of a function. We infer the size of the frame by considering the statements that manipulate SP in the function prologue. While we didn't observe any cases where the stack range was exceeded, this type of check forms the basis for further, more sophisticated improvements.

2.6 Malloc-based Security Outcomes

3 Experimental Setup

Our experimental setup consisted of running our plugins with BAP `0.9.6` [2] on a set of binaries, including GNU `Coreutils` and our own examples.

3.1 Unchecked Malloc

3.2 Stack Check Plugin

The stack check plugin was tested against the GNU `Coreutils` suite. We primarily chose the `Coreutils` suite since a) it can be compiled for ARM, and b) it contains numerous calls to `strcpy`, `memcpy`, `malloc`, and so forth. The stack check plugin, while configurable, concretely checked for function calls to:

- `strcat`
- `strcpy`
- `strncpy`
- `memcpy`

The plugin collected a number of statistics on the binaries, including the number of argument sites (that is, statements that set up registers before a call), the number of functions which contained calls to 'dangerous' functions, and the maximum number of statements on which an argument site statement was dependent. These are illustreated in Figure 1.

Figure 3.2 presents the total number of statements in a binary which affect unique argument sites.

The figures illustrate some interesting properties with respect to argument sites and data dependencies. Consider that each potentially dangerous function takes 2 or 3 parameters–of these, our data dependency plugin determined the maximum number of statements a single argument site statement is dependent on. We observe from Figure 1 that this could number in the hundreds, whereas the total number could range in the thousands (Figure 3.2). The implication here is that any additional reasoning and refinement to static analysis methods across cases composed of sizeable dependencies need to be carefully considered. For instance, if we add forms of pointer analysis, we may speculate about the scale of complexity of memory operations for argument sites.

We did not observe any violations with respect to the security outcomes in §2.5. This is perhaps unsurprising, given the ubiquitous use of the `Coreutils` suite. At the same time, it is difficult to choose a test suite which purposefully exhibits the stack properties that we check; see §5 for further discussion.

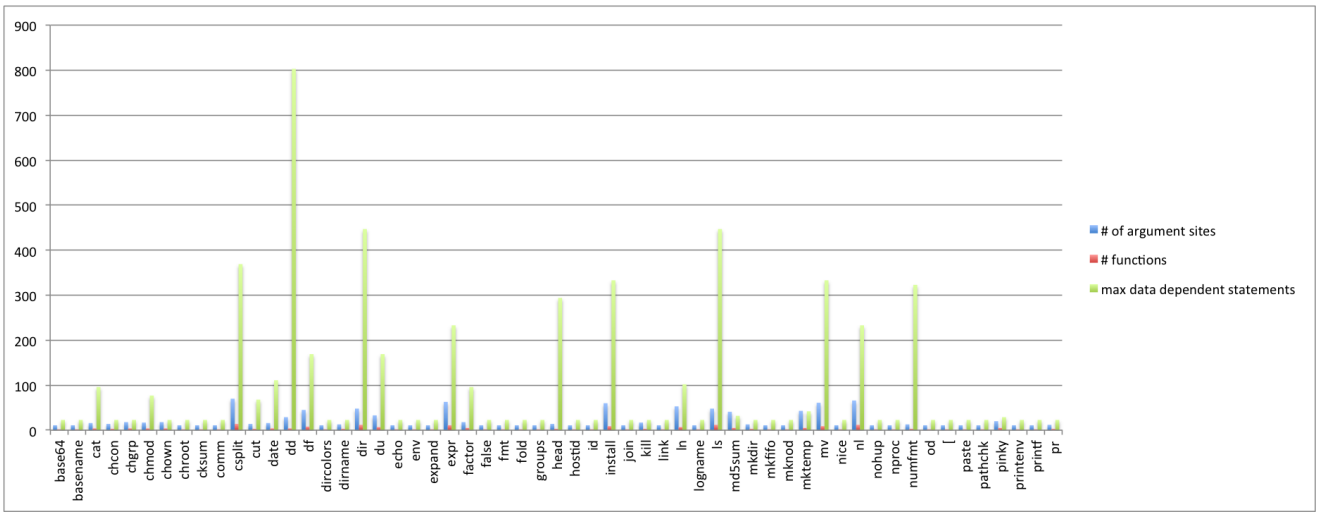
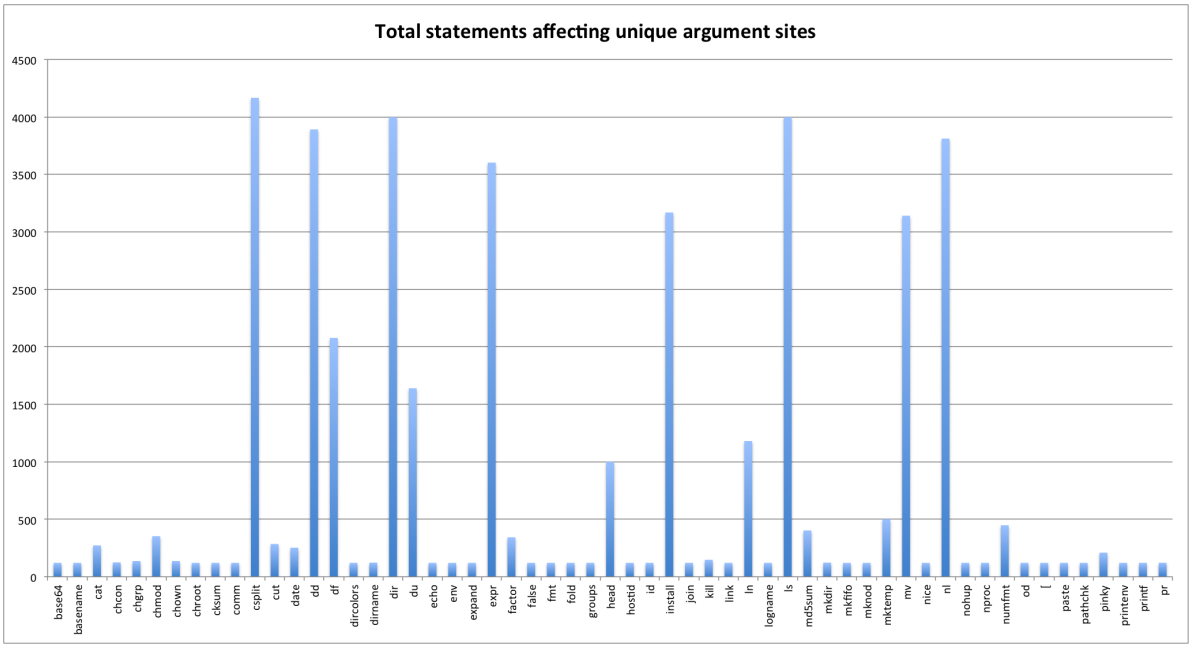


Figure 1: Coreutils statistics generated by plugin



3.3 Malloc

4 Experimental Evaluation

In general, we were pleased that our plugins ran successfully on our own example binaries as well as `Coreutils`. A key aspect of evaluating the success of our project is that we were able to start with a basic API in BAP, and after implementing a non-trivial amount of 'infrastructure' code, arrive at three individual plugins which are able to produce security outcomes.

Our evaluation of these plugins are largely qualitative rather than quantitative at this stage. However, they serve as very useful data points toward developing more sophisticated analyses.

4.1 Malloc Check Plugin

4.2 Stack Check Plugin

As mentioned, our stack check plugin did not trigger any violations when run over the `Coreutils` test suite. This result was largely anticipated: the `Coreutils` library was chosen so that we could verify that our plugins run and to rule out the possibility that binaries in `Coreutils` exhibited rudimentary flaws.

In critique of this plugin, there are a number of stack-based errors that could occur despite our checks. For example, consider a stack frame containing multiple local buffers at runtime. It is feasible that a buggy `strcpy` call could copy bytes beyond the boundary of a single buffer into an adjacent buffer, while remaining within the stack bounds. Our plugin would not detect this case. Consider a further case where memory accesses to the stack are dependent on general purpose registers—here we cannot conclude whether the access is within range of the stack frame.

A positive side-effect of the plugin is the potential of reusability, and by extension, refinement toward more sophisticated analyses. Recall that this plugin collects and makes use of statements that an argument site depends on. Thus, we have the ability to store the output of our analysis without having to rerun the plugin. For example, we may be able to couple this data with pointer-analysis, and infer stack boundaries.

4.3 Malloc Plugin

5 Surprises and Lessons Learned

There were a number of surprises and lessons learned throughout this project.

5.1 Reasoning about buffer overflows

An initial approach towards ensuring stack buffer overflow detection was to attempt to reason about sizes of buffers which are passed to "Potentially Dangerous Functions" [6] such as `strcpy`. This turned out to be an unfruitful approach for the scope of the project, since one would have to invariably have to reason about memory accesses.

For one, due to time constraints and complexity, we opted for an approach that does not require pointer analysis.

refinement as optimization

Mention why it's difficult: libxkbfile.so.1.0.2

challenging task... forumalting and putting into practice, concrete

Therefore, rather that attempting to infer and compare the sizes of buffers,...

For example, data dependency instructions numbering in the hundreds.

Rules out some possiblities. Requires continued refinement.

undecidability of ptr analysis.

Not strictly optimization, it is a refining of an analysis.

limitations highlight areas of refinement, not bad.

Deductive on what we need, towards an outcome (approach seems tow ork, with respect to an outcome!)

How did we respond

results were not surprising. but promising... what did you learn from this experience in general?

6 Conclusions and Future Work

Just as valuable to find out what works as what doesn't... that really is a meaningful contribution.
Further applications: nevernote, fgets, coverity?

Influence of statements...

7 Distribution of Total Credit

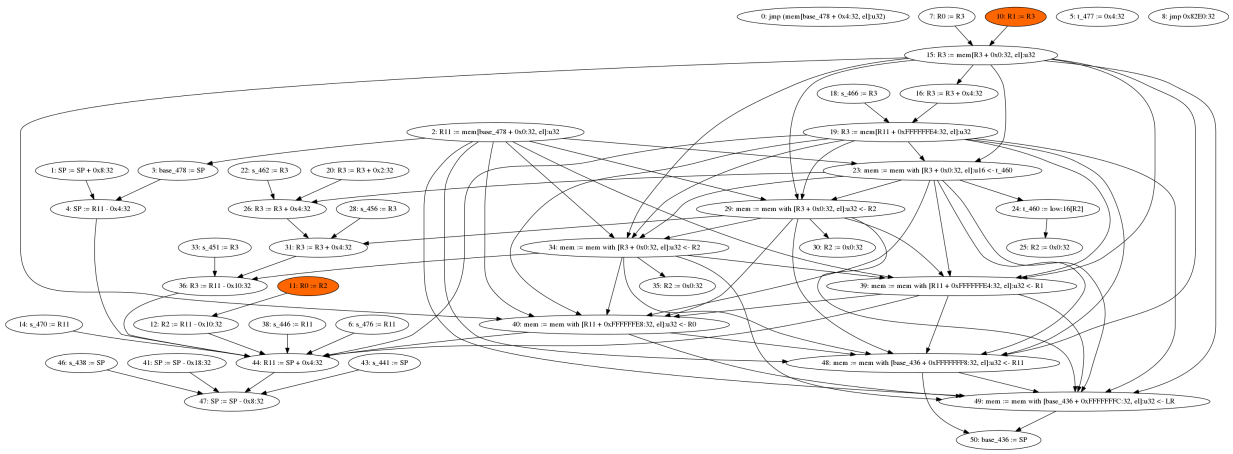
A Appendix

BIL Dissasembly of Listing 3

```
SP := SP - 0x8:32
mem := mem         with [base_436 + 0xFFFFFFF8:32, el]:u32 <- R11
mem := mem         with [base_436 + 0xFFFFFFFC:32, el]:u32 <- LR
base_436 := SP
R11 := SP + 0x4:32
t_439 := 0x4:32
s_438 := SP
SP := SP - 0x18:32
t_442 := 0x18:32
s_441 := SP
mem := mem         with [R11 + 0xFFFFFFE8:32, el]:u32 <- R0
mem := mem         with [R11 + 0xFFFFFFE4:32, el]:u32 <- R1
R3 := R11 - 0x10:32
t_447 := 0x10:32
s_446 := R11
R2 := 0x0:32
mem := mem         with [R3 + 0x0:32, el]:u32 <- R2
R3 := R3 + 0x4:32
t_452 := 0x4:32
s_451 := R3
R2 := 0x0:32
mem := mem         with [R3 + 0x0:32, el]:u32 <- R2
R3 := R3 + 0x4:32
t_457 := 0x4:32
s_456 := R3
R2 := 0x0:32
mem := mem         with [R3 + 0x0:32, el]:u16 <- t_460
t_460 := low:16[R2]
R3 := R3 + 0x2:32
t_463 := 0x2:32
s_462 := R3
R3 := mem[R11 + 0xFFFFFFE4:32, el]:u32
R3 := R3 + 0x4:32
t_467 := 0x4:32
s_466 := R3
R3 := mem[R3 + 0x0:32, el]:u32
R2 := R11 - 0x10:32
t_471 := 0x10:32
s_470 := R11
R0 := R2
R1 := R3
jmp 0x82E0:32
LR := 0x849C:32
```

BIL data dependency instructions for `argv[1]`

```
<(0x8444:32, 0)> base_436 := SP
<(0x8444:32, 1)> mem := mem         with [base_436 + 0xFFFFFFFC:32, el]:u32 <- LR
<(0x8444:32, 2)> mem := mem         with [base_436 + 0xFFFFFFF8:32, el]:u32 <- R11
<(0x8444:32, 3)> SP := SP - 0x8:32
<(0x8448:32, 2)> R11 := SP + 0x4:32
<(0x8450:32, 0)> mem := mem         with [R11 + 0xFFFFFFE8:32, el]:u32 <- R0
<(0x8454:32, 0)> mem := mem         with [R11 + 0xFFFFFFE4:32, el]:u32 <- R1
<(0x8458:32, 2)> R3 := R11 - 0x10:32
<(0x845C:32, 0)> R2 := 0x0:32
<(0x8460:32, 0)> mem := mem         with [R3 + 0x0:32, el]:u32 <- R2
<(0x8464:32, 2)> R3 := R3 + 0x4:32
<(0x8468:32, 0)> R2 := 0x0:32
<(0x846C:32, 0)> mem := mem         with [R3 + 0x0:32, el]:u32 <- R2
<(0x8470:32, 2)> R3 := R3 + 0x4:32
<(0x8474:32, 0)> R2 := 0x0:32
<(0x8478:32, 0)> t_460 := low:16[R2]
<(0x8478:32, 1)> mem := mem         with [R3 + 0x0:32, el]:u16 <- t_460
<(0x8480:32, 0)> R3 := mem[R11 + 0xFFFFFFE4:32, el]:u32
<(0x8484:32, 2)> R3 := R3 + 0x4:32
<(0x8488:32, 0)> R3 := mem[R3 + 0x0:32, el]:u32
```



Graphical output of BIL data dependencies, with highlighted arguments

References

[1]

[2] Binary analysis platform. `https://github.com/BinaryAnalysisPlatform/bap`. Accessed 03/16/2015.

[3] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: a binary analysis platform. In *Computer aided verification* (2011), Springer, pp. 463–469.

[4] Engler, D. Finding bugs with system-specific static analysis. `http://web.stanford.edu/~engler/paste02-talk.pdf`. Accessed 03/16/2015.

[5] Hallem, S., Chelf, B., Xie, Y., and Engler, D. *A system and language for building system-specific, static analyses*, vol. 37. ACM, 2002.

[6] Seacord, R. C. *The CERT C secure coding standard*. Pearson Education, 2008.