Project Report
LSP-305

A Taint Semantic Map to Accelerate Exploitation: FY20 Cyber Security Line Supported Program

T.R. Leek
A.T. Davis
A.J. King

7 March 2021

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

© 2020 MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

Massachusetts Institute of Technology
Lincoln Laboratory

A Taint Semantic Map to Accelerate Exploitation: FY20 Cyber Security Line Supported Program

T.R. Leek
A.T. Davis
A.J. King

Group 59

Project Report LSP-305

7 March 2021

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

This report describes and provides preliminary results for an analysis intended to accelerate the process of exploiting software vulnerabilities. The Taint Semantic Map (TSM) helps answer a critical question frequently posed by reverse engineers engaged in exploitation: what is the type and semantics of data in memory at a given point in program execution. This intelligence aids in exploring the implications of a read or write buffer overflow, as it tells one *what can I read or corrupt*? But it should also aid other aspects of exploitation, such as heap grooming and dynamic data structure understanding. The TSM is an application of dynamic taint analysis, in which memory access instructions are augmented to apply persistent labels to memory indicating the code that created or consumed that data. At a given point in execution, querying memory to determine the pattern of labels present can be used to determine type and semantics. We implement TSM in PANDA, the Platform for Architecture-Neutral Dynamic Analysis. In this report we motivate and describe TSM and its implementation. This is followed by some evaluations of its abilities and performance, including some sample maps from vulnerabilities seeded in programs.

This page intentionally left blank.

TABLE OF CONTENTS

		Page
	Abstract	iii
	List of Figures	vii
	List of Tables	ix
1.	OVERVIEW	1
2.	VULNERABILITY CONSTRAINTS	3
3.	VULNERABILITY OPPORTUNITIES	5
	3.1 Taint sources	5
	3.2 Taint queries, sinks	7
4.	IMPLEMENTATION	9
	4.1 Dynamic taint analysis	9
	4.2 Operating system introspection	9
	4.3 Computing the TSM	10
5.	EVALUATION	11
	5.1 The TSM for an injected bug	11
	5.2 Performance	11
6.	SUMMARY	15

This page intentionally left blank.

LIST OF FIGURES

Figure No.		Page
1	TSM excerpt. Thread/pc have been resolved, here, using PANDA introspection, to library and offset. Function name was determined using addr2line.	6
2	TSM excerpt, with data used as pointers and in branches labeled.	8
3	TSM excerpt for xmllint, output just before the read overflow occurred. The register rbx contains the pointer that can be corrupted and there are pointer and conditional data both above and below it in the map.	12
4	PANDA analysis time (left) and memory (right) vs instr count. Later instructions take longer and more memory to analyze.	13
5	Histogram of the number of instructions between a taint source (store instruction) and a taint sink (a ptr use or conditional). The mean is about 3.7m instructions.	14

LIST OF TABLES

Table No.		Page
1	Time and memory cost for computing the TSM, with breakdown of individual plugin costs. The abbreviations a, ll, tsm, tls, and tb refer, respectively, are glossed in the main discussion.	13

This page intentionally left blank.

1. OVERVIEW

Exploiting modern software is difficult and time consuming. The process can begin with a potential vulnerability, as indicated by an input that induces interesting behavior. That *interesting behavior* might be as nebulous as a program crash, in which case arduous *root cause determination* of the underlying vulnerability by a domain expert or reverse engineer will be necessary. Alternately, the interesting behavior may simply be the output of a dynamic detector or “sanitizer” that generates a report when certain kinds of vulnerabilities exhibit themselves. Sanitizers run a program under supervision, collecting side information as the program executes in order to detect vulnerabilities such as division-by-zero, buffer overflow, or use-after-free when they happen. Note that the output of a sanitizer, if precise, directly provides a determination of root cause. For example, when Clang’s Address Sanitizer (ASAN) detects a buffer overflow, it reports at the exact moment at which the vulnerability manifests itself. In this work, we restrict our attention to this case, in which a reasonably precise buffer overflow detector is available that can localize the vulnerability in time and space.

Given the interesting-behavior inducing input and the root cause vulnerability, the next steps towards exploitation are somewhat unclear. Consider the buffer overflow vulnerability, in which a program contains a bug that mean accesses to one of its internal buffers can be made to go out-of-bounds. If the overflow allows an attacker to *read* out-of-bounds, then she may obtain access to sensitive internal program data, perhaps even across the network, as in the Heartbleed bug. If the overflow allows an attacker to *write* out-of-bounds, then she may be able to change critical data and even get arbitrary code execution. But how, exactly, does she accomplish any of these goals? There are two problems the attacker must solve. The first involves determining the constraints upon the vulnerability. The attacker might have rather weak control over the address involved in the overflow, and, in the case of a write overflow some or no control over the value being written. The second problem is determining what opportunities a vulnerability such as an overflow provides. In the case of a buffer overflow, this is a matter of understanding what data is lying about in memory for unauthorized readout or corruption at the time the overflow occurs.

This page intentionally left blank.

2. VULNERABILITY CONSTRAINTS

We do not attempt to address the problem of determining constraints upon a vulnerability in this work. However, a few words are in order. Assuming an input that triggers a vulnerability such as a buffer overflow, it would seem useful to know the constraints upon that overflow. Does the attacker have complete control over a pointer, meaning she can set it to whatever address she likes? Or can she only add to it a number between 1 and 7? In the case of a write overflow, can the attacker choose, freely, what data is written? Or can she only write the 32-bit value 0x90909090?

If p is the pointer over which the attacker has enough control to cause an overflow, and v is the value being stored in the case of a write overflow, then

$$C(overflow) = C(p) \cup C(v) \tag{1}$$

is the set of constraints upon the overflow, which is the union of the constraints upon the pointer and those upon the value being written. Note that the overflow is known to occur for some input i, since a sanitizer observed it. However, it is unknown for how many other inputs I it will also manifest. For some programs, I might itself be infinite. For instance, it would be trivial to build a program for which *all inputs with 10 or more '(' characters in a row will trigger an overflow.* Further, note that many different paths through a program to the overflow point might all trigger an overflow. A known (but incomplete) solution to this problem is to mutate the input i to find more inputs that also trigger the same bug. The result is a set of inputs that can be used to gauge the constraints upon p and v. It is unclear how well this works, however, and it has not been studied. Determining the constraints upon a given vulnerability is a difficult and open problem.

This page intentionally left blank.

3. VULNERABILITY OPPORTUNITIES

It is the purpose of the TSM to address the problem of determining and exploring the opportunities a vulnerability presents. Given i and a sanitizer's report, we can know the exact moment of overflow, and what we require next is type and semantic information about the data we may be able to read or corrupt. Ideally, we would like a map we can read that tells us what kinds of data live at the addresses we are able to influence. Are they pointers? Are they critical control data? Are they data that will be used again by the program and how? The TSM can help elucidate this issue. It relies upon the intuition that, at least to some extent, the type and semantics of data in a program's memory are revealed by who created it, who reads it, and how it is used.

We use a a dynamic taint analysis to determine creator, reader, and use of data. A taint analysis affords three fundamental abilities.

1. The ability to affix labels to data in a program
2. The automatic propagation of those labels when data is copied or computed upon
3. The ability to query any program data at any point in time to determine if it has any labels

3.1 TAINT SOURCES

If we want to know who created some data, then we simply apply taint labels to it after a memory store operation occurs and arrange for the label to refer to the code that generated that data. To be more specific, the label we use at a store is, effectively, the triple $(thread, pc, t)$, where *thread* identifies the thread of execution[1], *pc* is the program counter which indicates what instruction in the code for that thread is executing, and t is time (for which we use an instruction count). We include time because we wish to differentiate between data written by the same code at different times. When we affix such a label to bytes in memory, we are stamping them with metadata that indicates what code generated them and when. Given this labeling, we have various options. One is to simply draw the map it affords us at critical points in the execution, by writing out, for each address near p, the creator of the data that resides there. An excerpt from one such map appears in Figure 1, and it contains a wealth of information. At address `0x7ffdb8146a68`, e.g., we have an 8-byte quantity generated by `libc`'s `malloc` function, which returns allocated heap data to the caller but also modifies and creates its own internal metadata, intended to be private. It is possible this quantity could be involved in exploitation; heap exploitation is an important modern technique. We can learn a lot about type and semantics if we consult the code indicated, offset `0x970f7` of the `malloc` library [2].

[1] our analysis is in PANDA which is whole-system, thus we need the thread
[2] mapping this offset to a line number in the source code is easy given debug symbols for the exact version of the library

```
Virtual Addresses
...
7ffdb8146a50[8]   libc:_IO_str_seekoff:942d4)
7ffdb8146a58[8]   libc:_IO_str_seekoff:942d2)
7ffdb8146a60[8]   libc:_IO_str_seekoff:942d0)
7ffdb8146a68[8]   libc:malloc:970f7)
7ffdb8146a70[8]   libxml:xmlRelaxNGValidateDoc:f6267)
7ffdb8146a78[8]   libxml:xmlRelaxNGValidateDoc:f6262)
7ffdb8146a80[8]   libxml:xmlRelaxNGValidateDoc:f6260)
7ffdb8146a88[8]   libc:_IO_str_seekoff:97071)
7ffdb8146a90[8]   libc:_IO_str_seekoff:97070)
7ffdb8146a98[8]   libxml:xmlNewNodeEatName:541bf)
7ffdb8146aa0[8]   libxml:xmlNewNodeEatName:541a6)
7ffdb8146aa8[8]   libxml:xmlNewNodeEatName:541a5)
7ffdb8146ab0[8]   libxml:xmlNewNodeEatName:541a3)
7ffdb8146ab8[8]   libxml:xmlNewDocNodeEatName:57cd9)
7ffdb8146ac0[8]   libc:_IO_str_seekoff:97070)
7ffdb8146ac8[8]   libxml:xmlNewDocNodeEatName:57cc8)
7ffdb8146ad0[8]   libxml:xmlAddChild:57019)
7ffdb8146ad8[8]   libxml:xmlAddChild:57015)
7ffdb8146ae0[8]   libxml:xmlAddChild:57013)
7ffdb8146ae8[8]   libxml:xmlSAX2StartElementNs:fa11b)
7ffdb8146af0[8]   libxml:xmlDictLookup:f6b22)
7ffdb8146af8[8]   libc:_IO_str_seekoff:9430e)
7ffdb8146b00[8]   libxml:inputPop:3add0)
7ffdb8146b08[8]   libc:_IO_str_seekoff:9456f)
7ffdb8146b10[8]   libc:_IO_str_seekoff:94534)
7ffdb8146b18[8]   libxml:inputPop:3adbe)
7ffdb8146b20[8]   libc:_IO_str_seekoff:94565)
7ffdb8146b28[8]   libxml:inputPop:3adac)
...
```

Figure 1. TSM excerpt. Thread/pc have been resolved, here, using PANDA introspection, to library and offset. Function name was determined using addr2line.

3.2 TAINT QUERIES, SINKS

We can learn more about the type and semantics of data if we query data in memory, programmatically, at strategic points, given some labeling of taint sources. In other words, we observe *how* tainted data is used at various *sinks* and use what we learn to augment the information in Figure 1.

Here are a number of ways we can employ taint queries to learn more about taint sources. The list is by no means exhaustive.

1. **Pointers**. We can arrange to query for taint on every pointer used in a load or store instruction. Taint labels discovered correspond to a taint source that originally stored a pointer.
2. **Conditionals**. We can query taint on the value used to decide a conditional branch or test. It may aid exploitation to know that the data stored by a taint source and available at a particular point is later used to decide program flow.
3. **Loop bounds**. Some subset of conditional branches will correspond to loop iteration. A simple static analysis that identifies branches that, when taken, result in looping should be able to distinguish these.
4. **System call arguments**. We can arrange to query taint on arguments to system calls. This could tell us that some taint source is really a file descriptor, a pointer to a socket, a file or directory name, etc.
5. **Function call arguments**. If we have prototypes for program or library function calls, then tainted arguments tell us about the type and semantics of the taint source.
6. **Function pointers**. If we query taint on a register being used in an indirect jump or call, then, if tainted, we learn that the taint source is a function pointer. Note that we may even know the prototype of that function since we know what function was actually called.
7. **Network output**. If we query taint on outgoing network packets and learn from what taint source it derives, then we can know buffers in memory that might serve as output for certain kinds of exploitation.
8. **Liveness**. We can know if a taint source is ever read/used again for a given input and program execution. Conversely, if it is never used this would be a weak indication the data may be dead and there is thus no way to exploit it.

We have implemented the first two of these taint sinks: pointers and conditionals. The additional information over Figure 1 can be seen in Figure 2, where we see that three of the 8-byte quantities in this portion of the TSM are pointers and two are used to decide branches. Note that these semantic labels are time-sensitive. That is, if we have a vulnerability such as a buffer overflow then it has to be triggerable *before* any of these semantic uses in order for it to be able to effect them.

```
Virtual Addresses
...
7ffdb8146a50[8]   libc:_IO_str_seekoff:942d4)          -- BR
7ffdb8146a58[8]   libc:_IO_str_seekoff:942d2)          -- PTR BR
7ffdb8146a60[8]   libc:_IO_str_seekoff:942d0)
7ffdb8146a68[8]   libc:malloc:970f7)
7ffdb8146a70[8]   libxml:xmlRelaxNGValidateDoc:f6267)
7ffdb8146a78[8]   libxml:xmlRelaxNGValidateDoc:f6262)   -- PTR
7ffdb8146a80[8]   libxml:xmlRelaxNGValidateDoc:f6260)
7ffdb8146a88[8]   libc:_IO_str_seekoff:97071)
7ffdb8146a90[8]   libc:_IO_str_seekoff:97070)
7ffdb8146a98[8]   libxml:xmlNewNodeEatName:541bf)
7ffdb8146aa0[8]   libxml:xmlNewNodeEatName:541a6)
7ffdb8146aa8[8]   libxml:xmlNewNodeEatName:541a5)
7ffdb8146ab0[8]   libxml:xmlNewNodeEatName:541a3)
7ffdb8146ab8[8]   libxml:xmlNewDocNodeEatName:57cd9)
7ffdb8146ac0[8]   libc:_IO_str_seekoff:97070)
7ffdb8146ac8[8]   libxml:xmlNewDocNodeEatName:57cc8)
7ffdb8146ad0[8]   libxml:xmlAddChild:57019)
7ffdb8146ad8[8]   libxml:xmlAddChild:57015)
7ffdb8146ae0[8]   libxml:xmlAddChild:57013)
7ffdb8146ae8[8]   libxml:xmlSAX2StartElementNs:fa11b)
7ffdb8146af0[8]   libxml:xmlDictLookup:f6b22)          -- PTR
7ffdb8146af8[8]   libc:_IO_str_seekoff:9430e)
7ffdb8146b00[8]   libxml:inputPop:3add0)
7ffdb8146b08[8]   libc:_IO_str_seekoff:9456f)
7ffdb8146b10[8]   libc:_IO_str_seekoff:94534)
7ffdb8146b18[8]   libxml:inputPop:3adbe)
7ffdb8146b20[8]   libc:_IO_str_seekoff:94565)
7ffdb8146b28[8]   libxml:inputPop:3adac)
...
```

Figure 2. TSM excerpt, with data used as pointers and in branches labeled.

4. IMPLEMENTATION

4.1 DYNAMIC TAINT ANALYSIS

The TSM implementation consists of a set of PANDA plugins to implement the taint sources and sinks described previously. TSM also required some substantial changes to PANDA's dynamic taint system. The PANDA TSM plugins output analytic results to the *pandalog*, a binary logging format employing Google's protocol buffers to serialize data structures. This pandalog is post-processed via a python script to generate textual output, as seen in Figures 1, 2, and 3.

TSM relies upon PANDA's existing dynamic taint analysis, which provides the abilities described at the beginning of Section 2,as well as a few additional features worth noting. PANDA is whole-system and it's taint analysis operates directly upon binaries. This means that multiple threads, library and even kernel code are all transparently subject to the same analysis. All machine instructions are tracked for taint, be they implemented in the underlying Qemu TCG intermediate language or in C helper functions. This is accomplished by translating TCG to LLVM's IR and using Clang to compile C helper functions to LLVM, after which the LLVM is augmented with taint transfer operations. PANDA distinguishes between taint transfers that are copies and those that are computation. Further, when there are multiple taint labels, PANDA keeps track of the *set of labels* associated with every byte in memory and in every register. Together, these latter two features permit us to tell when data loaded from memory is later copied to a new address in memory unchanged.

PANDA facilitates whole-system dynamic analysis via a number of callbacks. TSM uses the `PANDA_CB_AFTER_STORE` callback to be able to apply taint labels as described in Section 3.1. In order to accomplish this, a defect in PANDA's taint system had first to be addressed involving *when* this callback runs. Previously, it would run just after the emulated code that performed the store instruction, which would then immediately be followed by the injected taint propagating instrumentation. Thus, if the callback were used to apply taint labels to stored data those would promptly be erased or replaced by the taint transfer. This semantic error in PANDA has now been remedied and the callback works as expected. The TSM sinks thus far implemented (see Section 3.2) employ callbacks on pointer load/store operations and branches. These callbacks are from within the taint system. They were implemented previously and placed there because it is there that the various kinds of load/store and conditional branches unify to a very small number of LLVM constructs. This is not ideal, and while it provides results which seem right, we are in the same semantic peril as we were with `PANDA_CB_AFTER_STORE`. We are looking into more holistic solutions that will make it less likely a PANDA developer will have difficulties. One possibility is to implement all such callbacks which ought to run before/after some class of instruction (load/store/branch/etc) with the `PANDA_CB_INSN_TRANSLATE` and `PANDA_CB_INSN_EXEC` PANDA callbacks. We have avoided this in the past for two reasons. First, good disassemblers capable of determining instruction classes (loads, stores, branches, etc) did not exist. But the Capstone disassembler may now be mature enough for these purposes. Second, the same semantic hurdle exists with respect to these instruction level callbacks and the taint system as did with `PANDA_CB_AFTER_STORE` mentioned above. However, it should be straightforward to correct the instruction level callbacks using the essentially the same remedy as was used to fix `PANDA_CB_AFTER_STORE`.

4.2 OPERATING SYSTEM INTROSPECTION

TSM also relies upon PANDA's operating system introspection to determine the thread of operation and obtain the memory map for a process. PANDA's `asidstory` and `loaded_libs` plugins provide this information. The `asidstory` plugin logs information about instruction intervals during which threads operate, also noting internal details such as create time and thread id. The TSM restricts its attention to a

particular named thread, as the expectation is that its output would be consulted from a reverse engineering tool such as IDA Pro or Ghidra, both of which are executable-focused. The `loaded_libs` plugin periodically consults PANDA's operating system introspection to determine what modules are mapped into memory and where for the currently executing thread, sending this to the pandalog for later use. This is what allows the TSM to be able to render taint source labels $(thread, pc, t)$ in terms of named modules and offsets as in Figure 1.

4.3 COMPUTING THE TSM

The workflow for assembling the TSM involves using PANDA with plugins to label taint sources and query sinks as well as to determine necessary operating system information as described in the previous section. The following PANDA plugins are used, many of which required at least some additional software engineering to support TSM.

- `asidstory` – thread info logged
- `loaded_libs` – memory map info logged
- `tsm` – applies taint labels to data at store instructions and logs these
- `tainted_ldst` – queries ptr use in loads and stores and logs taint observed
- `tainted_branch` – queries condition used to decide branches and logs taint observed

These results are post-processed by a python script operating upon the pandalog. The script determines the particulars for the thread of interest (tid and create time), and subsequently focuses its attention upon logged items related to this thread. In its first pass over the log, the script identifies when taint source labels are observed at pointer use and conditional branch sinks. In a second pass, these labels can be augmented with this additional semantic information: "later used as a pointer" or "later used to decide a condition." Finally, the TSM itself is assembled incrementally. Every time a taint source is created, a per-thread map is updated from a set of virtual addresses of the store instruction to these semantic labels. Thus, the map updates with each store instruction written to the pandalog, and is highly dynamic. This means the analyst must specify, somehow, the precise moment at which the TSM is to be viewed. This might be programmatically, e.g., when some program counter or register contents are observed, or could simply be at some whole system replay instruction count known to be of interest.

5. EVALUATION

Thus far, we have evaluated our implementation of the TSM on an injected vulnerability and assessed its performance in a variety of ways.

5.1 THE TSM FOR AN INJECTED BUG

The TSM is intended to inform exploitation. As a demonstration of the possible efficacy of the TSM, we added a buffer overflow vulnerability to `xmllint` and crafted an input to trigger it. The bug was added using the ideas if not the machinery of the LAVA system, which uses a taint analysis, with labels applied to program inputs, to identify likely places where bugs might be readily added to a program. The taint analysis tells us where the input flows and is used in a program and how, which we can easily translate into modifications that insert a bug. A known bug is necessary, here, since we can know exactly how to trigger it and thus exactly when to output the TSM. The injected bug is an attacker-controlled out-of-bounds read, and a portion of the map both above and below the overflow are displayed in Figure 3. There are a number of pointers and condition-determining variables near p in memory along with source information that should assist a reverse engineer in understanding well enough to decide if the vulnerability is exploitable.

5.2 PERFORMANCE

The pandalog from which the TSM is computed is generated by a constellation of plugins which depend upon a dynamic taint analysis. All of this computation is currently quite expensive. Consider the `xmllint` program. A replay of its (and the rest of the operating system's) activity takes PANDA 3.65 seconds, which accounts for about 13.7m instructions[3]. That same replay with all the plugins TSM requires as well as logging data takes 5121 seconds, which is a slowdown of more than 1400x. The breakdown of the cost of running these various plugins together and separately is given in Table 1, where a, ll, tsm, tls, and tb, refer, respectively, to the `asidstory`, `loaded_libs`, `tsm`, `tainted_ldst`, and `tainted_branch` plugins. The lion's share of the slowdown is due to the TSM plugin and the attendant dynamic taint analysis. This slowdown is larger than expected and will be investigated.

Plots of the wall-clock time and memory required to analyze this xmllint replay appear in Figure 4. The horizontal axis in both plots is the same, a linear count of instructions executed. Both time and memory required to execute instructions increases as more instructions are executed. This could be a natural by-product of a taint analysis in which more of memory accrues labels as time goes on, and computation compounds the issue. This situation will be investigated; the TSM gains no benefit from sets of labels becoming associated with memory and registers.

The post-processing python script is also slow, taking an additional 5720 seconds to generate Figure 3. We believe this can be sped up dramatically, and much of its computation can be performed by panda much more quickly during replay.

An early hypothesis about the TSM was that it ought to be quite efficient, as the expected number of instructions between a store instruction at a taint source and the subsequent query at a sink would typically be small, i.e., in the 10s to hundreds. We plotted this in Figure 5 as a histogram and observe no such pattern. The number of instructions elapsing between a value being stored in memory and its use (load, store, branch) has an average value of 3.8M, and a median of 3.9M instructions. This will also be investigated.

[3] Intel Xeon Platinum 8268 CPU @ 2.90GHz, 512GB RAM

```
Virtual Addresses
...
55d0cfe63f00[8]  libc:_IO_enable_locks:8ebd9
55d0cfe63f08[8]  libc:_IO_str_seekoff:950d9                         -- BR
55d0cfe63f10[8]  libxml:__xmlParserInputBufferCreateFilename:6195b -- BR
55d0cfe63f18[8]  libxml:__xmlParserInputBufferCreateFilename:6196a -- BR
55d0cfe63f20[8]  libxml:__xmlParserInputBufferCreateFilename:61972
55d0cfe63f28[8]  libxml:xmlAllocParserInputBuffer:617f3            -- BR
55d0cfe63f30[8]  libxml:xmlAllocParserInputBuffer:617d9            -- PTR  BR
55d0cfe63f38[8]  libxml:xmlAllocParserInputBuffer:61840
55d0cfe63f40[4]  libxml:xmlAllocParserInputBuffer:61825            -- BR
55d0cfe63f44[4]  libxml:xmlAllocParserInputBuffer:617c1            -- BR
55d0cfe63f48[8]  libxml:xmlAllocParserInputBuffer:6182c
55d0cfe63f58[8]  libc:_IO_str_seekoff:950d9                         -- BR
55d0cfe63f60[8]  libxml:xmlBufCreateSize:a9dbc                      -- PTR  BR
55d0cfe63f68[4]  libxml:xmlBufAddLen:aa426                          -- BR
55d0cfe63f6c[4]  libxml:xmlBufAddLen:aa422                          -- BR
55d0cfe63f70[4]  libxml:xmlBufSetAllocationScheme:a9f84             -- BR
55d0cfe63f78[8]  libxml:xmlBufCreateSize:a9dc4
55d0cfe63f80[8]  libxml:xmlBufAddLen:aa40d                          -- BR
55d0cfe63f88[8]  libxml:xmlBufCreateSize:a9da7                      -- BR
55d0cfe63f90[8]  libxml:xmlBufCreateSize:a9d89                      -- BR
55d0cfe63f98[4]  libxml:xmlBufCreateSize:a9d82                      -- BR
55d0cfe63fa8[8]  libc:_IO_str_seekoff:950d9                         -- BR
55d0cfe63fb0[1]  libxml:xmlBufCreateSize:a9dc1
55d0cfe64042*    rbx the pointer
55d0cfe647b0[16] libc:memcpy:bb6bf
55d0cfe647c0[16] libc:memcpy:bb696
...
55d0cfe64e1a[16] libc:memcpy:bb6b2
55d0cfe64e2a[16] libc:memcpy:bb6af
55d0cfe64e3a[1]  libxml:xmlBufAddLen:aa42e
55d0cfe65fb8[8]  libc:_IO_str_seekoff:950d9                         -- BR
55d0cfe65fc0[8]  libxml:xmlNewInputFromFile:36d3f                   -- PTR  BR
55d0cfe65fc8[8]  libxml:xmlNewInputFromFile:36d96                   -- BR
55d0cfe65fd0[8]  libxml:xmlNewInputFromFile:36dab
55d0cfe65fd8[8]  libxml:xmlBufResetInput:ab3cd                      -- BR
55d0cfe65fe0[8]  libxml:xmlParseVersionInfo:46ee2                   -- PTR  BR
55d0cfe65fe8[8]  libxml:xmlParserInputGrow:356c5                    -- BR
55d0cfe65ff0[4]  libxml:xmlNewInputStream:369a7
55d0cfe65ff4[4]  libxml:xmlParseCharData:3f34e                      -- BR
55d0cfe65ff8[4]  libxml:xmlParseVersionInfo:46edb
55d0cfe65ffc[4]  libxml:xmlNewInputStream:369a7
55d0cfe66000[8]  libxml:xmlNewInputStream:369a7                     -- BR
55d0cfe66008[8]  libxml:xmlNewInputStream:369a7
55d0cfe66010[8]  libxml:xmlNewInputStream:369a7
55d0cfe66018[8]  libxml:xmlNewInputStream:369a7
55d0cfe66020[4]  libxml:xmlNewInputStream:369b4
55d0cfe66024[4]  libxml:xmlNewInputStream:369d0                     -- BR
55d0cfe66028[8]  libc:_IO_str_seekoff:950d9                         -- PTR  BR
55d0cfe66030[8]  libxml:xmlDictLookup:f6d8d
```

Figure 3. TSM excerpt for xmllint, output just before the read overflow occurred. The register rbx contains the pointer that can be corrupted and there are pointer and conditional data both above and below it in the map.

Analysis	Time (sec)	Overhead (wrt none)	Memory GB
none	3.65	–	0.41
a	5.08	1.39x	0.41
ll	21.3	5.84x	0.54
tsm	4746	1300x	31.39
a,ll,tsm,tls,tb	5121	1404x	31.59

TABLE 1

Time and memory cost for computing the TSM, with breakdown of individual plugin costs. The abbreviations a, ll, tsm, tls, and tb refer, respectively, are glossed in the main discussion.

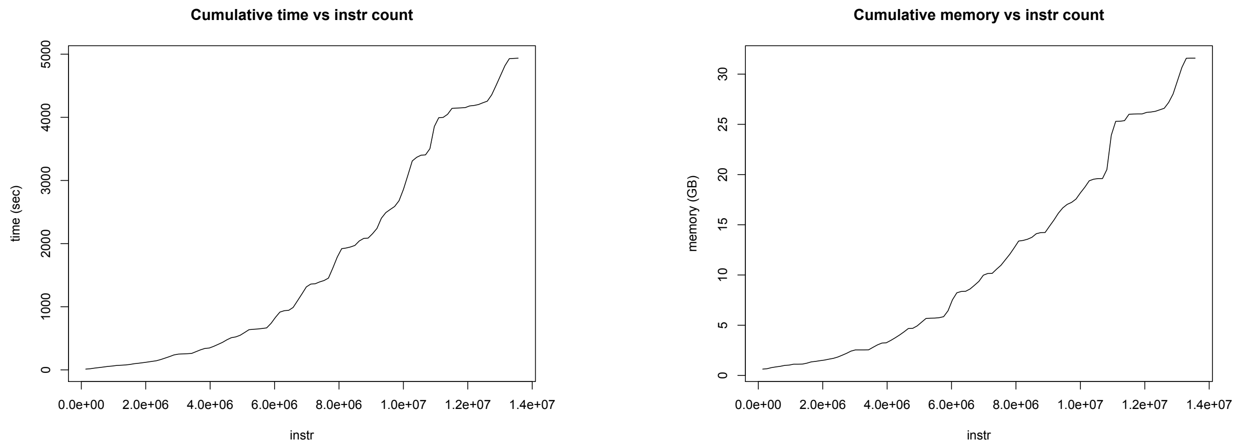

Figure 4. PANDA analysis time (left) and memory (right) vs instr count. Later instructions take longer and more memory to analyze.

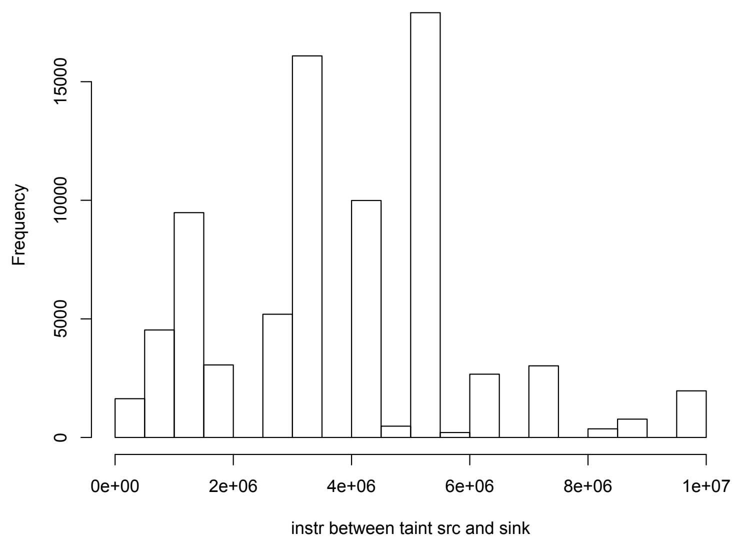

Figure 5. Histogram of the number of instructions between a taint source (store instruction) and a taint sink (a ptr use or conditional). The mean is about 3.7m instructions.

6. SUMMARY

The TSM prototype seems to provide useful information but a number of experiments are in order to give weight to this claim. We will investigate its usefulness in exploiting a real vulnerability, in addition to the injected one, and for a real program other than `xmllint`. We will also investigate its ability to eludicate the pattern of objects in memory during heap grooming or heap spray. Performance also needs to be addressed. Previous estimates of the slowdown due to invoking PANDA's taint system were less than 100x, as compared with the 1400x observed here. This will be investigated and we will pursue opportunities for optimization.