Taint Driven Crash Analysis

Hack In The Box Beijing 2018

Richard Johnson
rjohnson@fuzzing.io

whoami

- Richard Johnson
  Research Lead
  @richinseattle
  rjohnson@fuzzing.io

- Cisco Talos :: Vulndev
  - Third party vulnerability research
    - Microsoft
    - Apple
    - Oracle
    - Adobe
    - Google
    - IBM, HP, Intel
  - Security tool development
    - Fuzzers, Crash Triage
  - Mitigation development

Root Cause Analysis

- Execution Path
  - What code paths were executed
  - What parts of the execution interacted with external data
- Input Determination
  - Which input bytes influence the crash
- Root Cause
  - Which line of code needs to be patched

Common Vulnerability Analysis Scenarios

- Fuzzing
  - Spray ‘n Pray
  - Grammar-based
  - “Fuzzing with Code Fragments”
- Static Analysis
  - Intra-procedural Analysis
  - Manual code review
- Third Party
  - In-the-wild exploitation
  - Vulnerability response teams
  - Vulnerability brokers

Previous Tooling

- Execution Path
  - Process Stalker, CoverIt (hexblog), BlockCov, IDA PIN Block Trace
  - Bitblaze, Taintgrind, VDT
- Input Determination
  - delta, tmin, diff
- Exploitability
  - !exploitable
  - CrashWrangler
  - CERT Triage Tools

Automation Methods

- Execution Path
  - Code Coverage
  - Taint Analysis
- Input Determination
  - Slicing
- Exploitability
  - Symbolic Execution
  - Abstract Interpretation

Automation Methods

- Execution Path
  - Code Coverage
  - **Taint Analysis**
- Input Determination
  - **Slicing**

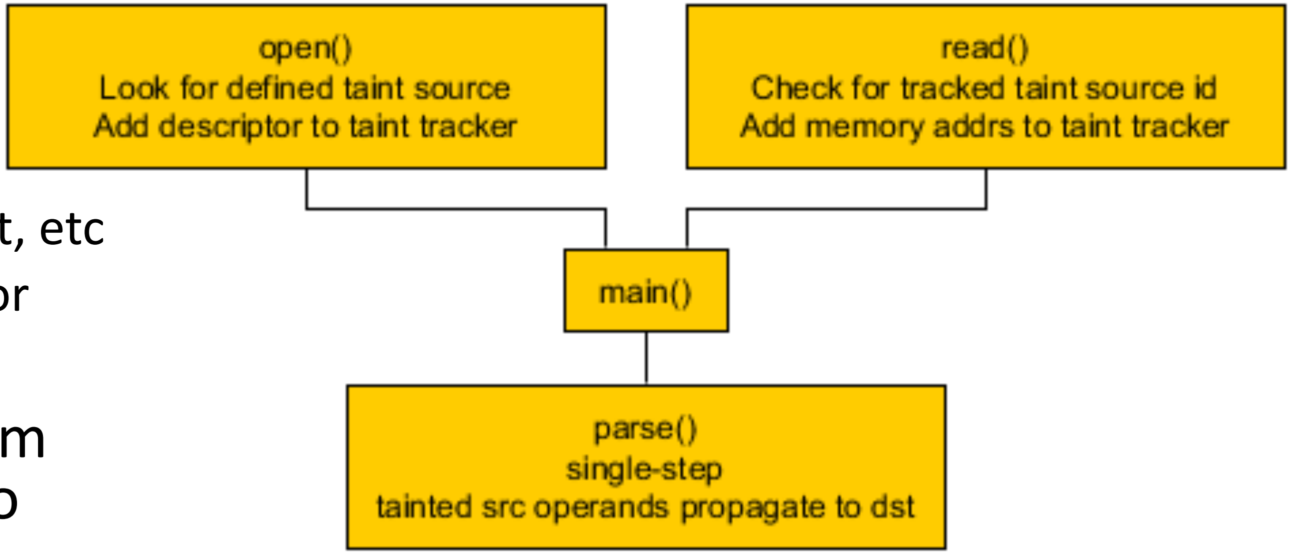
Taint Analysis

Concept

- Formally – Information Flow Analysis
  - Type of dataflow analysis
  - Can be static or dynamic, often hybrid
  - Applied to track user controlled data through execution
- Methodology
  - Define taint sources
  - Single-step execution
  - Apply taint propagation policy for each instruction
  - Apply taint checks (if any)

Concept

- Define Taint Sources
  - Hook I/O Functions
  - Look for taint sources
    - File name, network ip:port, etc
    - Track tainted file descriptor
  - Single-step
  - Add future data reads from taint source descriptors to the taint tracking engine
  - Apply taint policy on each instruction



Concept

- Define Taint Sources
  - Hook I/O Functions
  - Look for taint sources
    - File name, network ip:port, etc
    - Track tainted file descriptor
  - Single-step
  - Add future data reads from taint source descriptors to the taint tracking engine
  - Apply taint policy on each instruction

EXPLICIT TAINT PROPAGATION

```
A = TAINT()
B = A
C = B + 1
D = C * B
E = *(D)
```

IMPLICIT TAINT PROPAGATION

```
A = TAINT()
IF A > B:
    C = TRUE
ELSE:
    C = FALSE
```

Implementation Details

- We will utilize Intel PIN to perform dynamic binary translation to instrument a target binary for tracing
- Binary translation is a robust program modification technique
  - JIT for hardware ISAs
- PIN supplies a robust API and framework for binary instrumentation
  - Supports easily hooking I/O functions for taint sources
  - High performance single-stepping
  - Supports instrumenting at instruction level for taint propagation / checks

Implementation Details

- We need to look for user defined taint sources returned from system calls and then single step watching memory propagate throughout the program

- To achieve our taint tracing we forked the PIN tool from the Binary Analysis Platform from Carnegie-Mellon University
  - Worked with the authors of BAP since early 2012 to improve the tracer so it performs acceptably against complex COTS software targets on Windows
  - Added code coverage and memory dump collection

Implementation Details – Trace Format

- The log is saved as a custom binary format with embedded protobuf structures. We use ‘piqi’ to define the structures and compile to protobuf

```
root@d14e0cf39781:/moflow/bap/libtracewrap/libtrace/piqi-files# cat stdframe.piqi
.include [
         .module types
]

% Frame representing the execution of a single assembly instruction
.record [
```

Implementation Details – Trace Format

```
 % Address of the instruction
.field [
       .type address
       .code 1
]

% Thread id that executed the instruction
.field [
       .type thread-id
       .code 2
]
```

Implementation Details – Trace Format

```
% Raw bytes of the instruction
.field [
        .name rawbytes
        .type binary
        .code 3
]
```

Implementation Details – Trace Format

```
% Operands values read by the instruction
.field [
       .name operand-pre-list
       .type operand-value-list
       .code 4
]
```

Implementation Details – Trace Format

```
% Operands values written by the instruction
.field [
        .name operand-post-list
        .type operand-value-list
        .code 5
        .optional
]
```

Implementation Details

- Taint Propagation Policy
  - Tree of tainted references to registers and bytes of memory are individually tracked
  - If input operands contain taint, propagate to all output operands
  - No control flow tainting
  - Optionally taint index registers
    - All index registers for LEA instructions are tainted

  - No support for MMX, Floating point FCMOV, SSE PREFETCH

Design Considerations

- Taint Policy
  - Implicit Information Flows
    - Over-tainting
      - Most common when applying implicit taint via control flow
    - Under-tainting
      - If control flow taint is ignored
- Performance
  - Execution Speed
    - Analysis on each instruction is expensive
    - Avoid context switching
  - Memory Overhead

Taint Tracer Usage

root@d14e0cf39781:/moflow/slicer# ../bap/pin/pin -t ../tracer/gentrace32.so -help -- ./demo/demo

Pin tools switches

-bb_file [default]

    Store the set of visited BBs in a text file

-coverage_track [default false]

    Enable coverage tracking (only unique BBs)

-exn_file [default]

    Store info about exception in a text file

-follow_progs

    Follow the given program names if they are exec'd

Taint Tracer Usage

-freq [default 10000]

Report value of eip every n instructions.

-log_key_frames [default false]

Periodically output key frames containing important program values

-log_limit [default 0]

Number of instructions to limit logging to.

-log_syscalls [default false]

Log system calls (even those unrelated to taint)

Taint Tracer Usage

-logall_after [default false]

Log all (even untainted) instructions after the first tainted instruction

-logall_before [default false]

Log all (even untainted) instructions before and after the first tainted instruction

-logfile [default pintool.log]

The log file path and file name

-logone_after [default false]

Log the first instruction outside of the log range (taint-start/end), and then exit.

Taint Tracer Usage

-o [default out.bpt]

    Trace file to output to.

-skip_taints [default 0]

    Skip this many taint introductions

-snapshot_file [default]

    File name for memory snapshot

-stack_dump [default 0]

    How many bytes of stack to dump on exception

Taint Tracer Usage

-taint_args [default false]

Command-line arguments will be considered tainted

-taint_env

Environment variables to be considered tainted

-taint_files

Consider the given files as being tainted

-taint_indices [default false]

Values loaded with tainted memory indices will be considered tainted

-taint_net [default false]

Everything read from network sockets will be considered tainted

Taint Tracer Usage

-trig_count [default 0]

Number of times trigger will be executed before activating.

-trig_mod [default]

Module that trigger point is in.

-unique_logfile [default 0]

The log file names will contain the pid

-visited_bb_file [default]

Do not log hits to BBs listed in this file

Taint Tracer Usage

-trig_count [default 0]

Number of times trigger will be executed before activating.

-trig_mod [default]

Module that trigger point is in.

-unique_logfile [default 0]

The log file names will contain the pid

-visited_bb_file [default]

Do not log hits to BBs listed in this file

```
root@d14e0cf39781:/moflow/slicer# sudo ../bap/pin/pin -t ../tracer/gentrace32.so -taint
_indices -taint_files input.txt -snapshot-file /tmp/demo.snapshot -o /tmp/demo.trace --
 ./demo/demo tlv demo/input.txt 2>&1 | head -18
Logging initially enabled: 0
Code cache limit is 0
Starting program
This is modload()
This is modload()
Thread 0 starting
Not opening /etc/ld.so.cache
Not opening /lib32/libc.so.6
This is modload()
Opening tainted file: demo/input.txt
Tainting 1000 bytes from read at dda0a000, fd=5
Taint introduction #0. @dda0a000/1000 bytes: file demo/input.txt
adding new mapping from file demo/input.txt to 0 on taint num 1
adding new mapping from file demo/input.txt to 1 on taint num 2
adding new mapping from file demo/input.txt to 2 on taint num 3
adding new mapping from file demo/input.txt to 3 on taint num 4
adding new mapping from file demo/input.txt to 4 on taint num 5
adding new mapping from file demo/input.txt to 5 on taint num 6
```

```
adding new mapping from file demo/input.txt to ffd on taint num ffe
adding new mapping from file demo/input.txt to ffe on taint num fff
adding new mapping from file demo/input.txt to fff on taint num 1000
Activating taint analysis
First logged instruction
First tainted instruction
closed tainted fd 5
total_size: 65535, record_count 1111638594

 ----------- Tainted Regs ------------
ecx = ffffffff
eflags = ffffffff
ymm0 = ffffffff

 ----------- Tainted Mem ------------
Addr: 804b008 -> ffffffff
Addr: 804b009 -> ffffffff
Addr: 804b00a -> ffffffff
Addr: 804b00b -> ffffffff
Addr: 804b00c -> ffffffff
Addr: 804b00d -> ffffffff
```

- Addresses with a taint value of -1 have merged paths from multiple bytes from the user supplied input

- Positive integers correspond to byte offsets in the original input

- Non-tainted addresses would hold a 0 in that field so offsets are base 1, not base 0

```
Addr: dd540025 -> ffffffff
Addr: dd540026 -> ffffffff
Addr: dd540027 -> ffffffff
Addr: dd540028 -> ffffffff
Addr: dd540029 -> ffffffff
Addr: dda0a000 -> 1
Addr: dda0a001 -> 2
Addr: dda0a002 -> 3
Addr: dda0a003 -> 4
Addr: dda0a004 -> 5
Addr: dda0a005 -> 6
Addr: dda0a006 -> 7
Addr: dda0a007 -> 8
Addr: dda0a008 -> 9
```

- When an exception occurs the crash context is saved along with memory
- Each allocated page is written to disk for later automated crash analysis

```
edx v=0x0804b010, t=0x00000000
esi v=0x00000000, t=0x00000000
edi v=0x00000d80, t=0x00000000
ebp v=0xffffd238, t=0x00000000
esp v=0xffffd1f4, t=0x00000000
eflags v=0x00010202, t=0xffffffff
eip v=0xddba7cdf, t=0x00000000
0x01048000 - 0x017a3000 (7532KB)
0x017a6000 - 0x017a7000 (4KB)
0x017aa000 - 0x017d4000 (168KB)
0x017d5000 - 0x0181c000 (284KB)
0x08048000 - 0x0806c000 (144KB)
0xdd0fc000 - 0xdd1fe000 (1032KB)
0xdd20e000 - 0xdd20f000 (4KB)
0xdd239000 - 0xdd23a000 (4KB)
0xdd26d000 - 0xdf1e0000 (32204KB)
0xdf1e1000 - 0xdf5fb000 (4200KB)
0xdf5fc000 - 0xf7ffe000 (403464KB)
0xff7fe000 - 0xffffe000 (8192KB)
Total snapshot size: 468205568B (457232KB) (446MB)
Received fatal signal b
```

```
.text:08048871
.text:08048872
.text:08048872 ; =============== S U B R O U T I N E =======================================
.text:08048872
.text:08048872
.text:08048872 foo             proc near               ; CODE XREF: nice_crash↑p
.text:08048872
.text:08048872 arg_4           = dword ptr  8
.text:08048872
.text:08048872                 mov     esi, [esp+arg_4]
.text:08048876                 xor     eax, eax
.text:08048878                 lodsb                   ;
.text:08048878                                         ;   @context "R_EAX" = 0x0, 0, u32, wr @context "R_ESI" = 0x9cb0000, 0, u32, rd
.text:08048878                                         ;   @context "EFLAGS" = 0x246, 0, u32, rd
.text:08048878                                         ;   @context "mem[0x9cb0000]" = 0x41, 1, u8, rd
.text:08048878                                         ; label pc_0x8048878
.text:08048878                                         ; T_32t0:u32 = R_DFLAG:u32
.text:08048878                                         ; T_32t1:u32 = R_ESI:u32
.text:08048878                                         ; T_8t2:u8 = mem:?u32[T_32t1:u32, e_little]:u8
.text:08048878                                         ; R_EAX:u32 = R_EAX:u32 & 0xffffff00:u32 | pad:u32(T_8t2:u8)
.text:08048878                                         ; T_32t3:u32 = T_32t1:u32 + T_32t0:u32
.text:08048878                                         ; R_ESI:u32 = T_32t3:u32
.text:08048878                                         ;
.text:08048879                 xor     edi, edi
.text:0804887B                 add     edi, eax        ;
.text:0804887B                                         ;   @context "R_EDI" = 0x0, 0, u32, rw
.text:0804887B                                         ;   @context "R_EAX" = 0x41, 1, u32, rd @context "EFLAGS" = 0x246, 0, u32, wr
.text:0804887B                                         ; label pc_0x804887b
.text:0804887B                                         ; T_t1:u32 = R_EDI:u32
.text:0804887B                                         ; T_t2:u32 = R_EAX:u32
.text:0804887B                                         ; R_EDI:u32 = R_EDI:u32 + T_t2:u32
.text:0804887B                                         ; R_CF:bool = R_EDI:u32 < T_t1:u32
.text:0804887B                                         ; R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EDI:u32 ^ T_t1:u32 ^ T_t2:u32))
.text:0804887B                                         ; R_OF:bool = high:bool((T_t1:u32 ^ ~T_t2:u32) & (T_t1:u32 ^ R_EDI:u32))
.text:0804887B                                         ; R_PF:bool =
.text:0804887B                                         ;   ~low:bool(R_EDI:u32 >> 7:u32 ^ R_EDI:u32 >> 6:u32 ^ R_EDI:u32 >> 5:u32 ^
.text:0804887B                                         ;             R_EDI:u32 >> 4:u32 ^ R_EDI:u32 >> 3:u32 ^ R_EDI:u32 >> 2:u32 ^
.text:0804887B                                         ;             R_EDI:u32 >> 1:u32 ^ R_EDI:u32)
.text:0804887B                                         ; R_SF:bool = high:bool(R_EDI:u32)
.text:0804887B                                         ; R_ZF:bool = 0:u32 == R_EDI:u32
.text:0804887B                                         ;
.text:0804887D                 sub     edi, 30h        ;
.text:0804887D                                         ;   @context "R_EDI" = 0x41, 1, u32, rw
.text:0804887D                                         ;   @context "EFLAGS" = 0x206, 1, u32, wr
.text:0804887D                                         ; label pc_0x804887d
.text:0804887D                                         ; T_t:u32 = R_EDI:u32
.text:0804887D                                         ; R_EDI:u32 = R_EDI:u32 - 0x30:u32
.text:0804887D                                         ; R_CF:bool = T_t:u32 < 0x30:u32
.text:0804887D                                         ; R_OF:bool = high:bool((T_t:u32 ^ 0x30:u32) & (T_t:u32 ^ R_EDI:u32))
.text:0804887D                                         ; R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EDI:u32 ^ T_t:u32 ^ 0x30:u32))
.text:0804887D                                         ; R_PF:bool =
.text:0804887D                                         ;   ~low:bool(R_EDI:u32 >> 7:u32 ^ R_EDI:u32 >> 6:u32 ^ R_EDI:u32 >> 5:u32 ^
.text:0804887D                                         ;             R_EDI:u32 >> 4:u32 ^ R_EDI:u32 >> 3:u32 ^ R_EDI:u32 >> 2:u32 ^
.text:0804887D                                         ;             R_EDI:u32 >> 1:u32 ^ R_EDI:u32)
.text:0804887D                                         ; R_SF:bool = high:bool(R_EDI:u32)
.text:0804887D                                         ; R_ZF:bool = 0:u32 == R_EDI:u32
.text:0804887D                                         ;
```

Trace Slicing

Taint Slicing for Root Cause Analysis

- Now we have collected all instructions that interacted with user data, the values of that data for each instruction, and a snapshot of memory at a crash

- We will construct a dataflow dependency graph that holds all relationships between the instructions and the memory values through out the lifetime of the program

- Finally we will select a byte on that graph and find the path from exception back to the original input offset and value

Concept

- Methodology
  - Collect trace
  - Convert trace to BAP IL
  - Select location and value of interest (register or memory address)
  - Select direction of slice
  - Follow dependencies in desired direction to produce sub-graph

Concept

- Trace slicing finds the sub-graph of dependencies between two nodes
  - All nodes that influence or are influenced by specified node can be isolated
  - Reachability Problem
- Forward Slicing
  - Slice forward to determine instructions influenced by selected value
- Backward Slicing
  - Slice backward to locate the instructions influencing a value
  - Collect constraints to determine the degree of control over the value

Forward Slicing

- Slice forward to determine instructions influenced by a value

```
S = {v}
For each stmt in statements:
    If vars(stmt.rhs) ∩ S != ∅ then
        S := S ∪ {stmt.lhs}
Return S
```

stmt	S
el_size, el_count, el_data = read()	{**el_size**}
total_size = **el_size** * el_count	{**el_size**, **total_size**}
buf = malloc(**total_size**)	{el_size, **total_size**}
while count < el_count	{el_size, total_size}
offset = count * **el_size**	{**el_size**, total_size, **offset**}
data_offset = el_data + **offset**	{el_size, total_size, **offset**, **data_offset**}
buf_offset = buf + **offset**	{el_size, total_size, **offset**, data_offset, **buf_offset**}
memcpy(buf_offset, **data_offset**, **el_size**)	{**el_size**, total_size, offset, data_offset, **buf_offset**}

Backward Slicing

- Slice backward to locate the instructions influencing a value

```
S = {v}
For each stmt in reverse(statements):
    If {stmt.lhs} ∩ S != ∅ then
        S := S ∪ vars(stmt.rhs)
Return S
```

stmt	S
`el_size, el_count, el_data = read()`	`{data_offset, el_data, offset, count, el_size}`
`total_size = el_size * el_count`	`{data_offset, el_data, offset, count, el_size}`
`buf = malloc(total_size)`	`{data_offset, el_data, offset, count, el_size}`
`while count < el_count`	`{data_offset, el_data, offset, count, el_size}`
`    offset = count * el_size`	`{data_offset, el_data, offset, count, el_size}`
`    data_offset = el_data + offset`	`{data_offset, el_data, offset}`
`    buf_offset = buf + offset`	`{data_offset}`
`    memcpy(buf_offset, data_offset, el_size)`	`{data_offset}`

- To perform slicing on native assembly language we need to understand the semantics of every instruction
- This is tedious and error prone, especially when support for multiple architectures is desired
- A common solution for this problem is to use an intermediate assembly language that expands complex instructions to simplified RISC like architecture with all side effects explicit

Implementation Details

- BAP includes an intermediate assembly language definition called BIL
- BIL expands each native assembly instruction into a sequence of instructions representing each side effect
- Each instruction is easier to analyze and side effects are explicit
- We only have to handle assignments of the form *var := exp*

program	::=	*stmt**
stmt	::=	*var* := *exp* | `jmp`(*exp*) | `cjmp`(*exp,exp,exp*)
		| `halt`(*exp*) | `assert`(*exp*) | `label` *label_kind*
		| `special`(string)

Implementation Details

- BAP includes an intermediate assembly language definition called BIL
- BIL expands each native assembly instruction into a sequence of instructions representing each side effect
- Each instruction is easier to analyze and side effects are explicit
- We only have to handle assignments of the form *var := exp*

```
.text:08048887  mov edx, [edi+11223344h] ;
.text:08048887    ;   @context "R_EDX" = 0x1000, 0, u32, wr
.text:08048887    ;   @context "R_EDI" = 0x11, 1, u32, rd
.text:08048887    ;   @context "mem[0x11223355]" = 0x0, 0, u8, rd
.text:08048887    ;   @context "mem[0x11223356]" = 0x0, 0, u8, rd
.text:08048887    ;   @context "mem[0x11223357]" = 0x0, 0, u8, rd
.text:08048887    ;   @context "mem[0x11223358]" = 0x0, 0, u8, rd
.text:08048887    ; label pc_0x8048887
.text:08048887    ; R_EDX:u32 = mem:?u32[R_EDI:u32 + 0x11223344:u32, e_little]:u32
```

Slicing with moflow

- Get the prebuilt docker image:

```
vulndev@vulndev-x64 ~ $ sudo docker run -ti moflow/moflow-0.8
[sudo] password for vulndev:
root@d14e0cf39781:/moflow# cd slicer/
root@d14e0cf39781:/moflow/slicer# ls
common.cmi  ilprint.sh     readme.txt   slicer.cmo            triage.sh
common.cmo  makefile       run_demo.sh  slicer.ml
common.ml   motriage.ml    slicer       snapshot.format.txt
demo        prep-slice.sh  slicer.cmi   tests
root@d14e0cf39781:/moflow/slicer# ./run_demo.sh
```

Slicing with moflow

- First we will record the taint trace and then we will convert to the BAP intermediate language.

- We will use concrete substitution (concretization) to load the values that were in memory into the IL

- BAP is capable of doing fully static analysis but for our purposes of crash analysis we want to use the memory values

Slicing with moflow

- First we will record the taint trace and then we will convert to the BAP intermediate language.

- We will use concrete substitution (concretization) to load the values that were in memory into the IL

- BAP is capable of doing fully static analysis but for our purposes of crash analysis we want to use the memory values

Slicing with moflow

- root@d14e0cf39781:/moflow/slicer# sudo ../bap/pin/pin -t ../tracer/gentrace32.so -taint_indices -taint_files input.txt -snapshot-file /tmp/demo.snapshot -o /tmp/demo.trace -- ./demo/demo tlv demo/input.txt 2>/dev/null
- total_size: 65535, record_count 1111638594

- root@d14e0cf39781:/moflow/slicer# ../utils/iltrans -trace /tmp/demo.trace -trace-concrete-subst -trace-dsa -pp-ast /tmp/demo.trace.il
- Concrete Substitution Run: Done! (0.412670 seconds)

Slicing with moflow

- We now have a file in /tmp/demo.trace.il that is plaintext. We could have output bjson, json, protobuf, or plaintext

- We see taint introduced at the top

```
/*ReadSyscall*/ @taint_intro 1, "file demo/input.txt", 0
 @context "mem32[0xdda0a000]" = 0x41, 1, u8, wr
 @taint_intro 2, "file demo/input.txt", 1
 @context "mem32[0xdda0a001]" = 0x41, 2, u8, wr
 @taint_intro 3, "file demo/input.txt", 2
 @context "mem32[0xdda0a002]" = 0x41, 3, u8, wr
 @taint_intro 4, "file demo/input.txt", 3
 @context "mem32[0xdda0a003]" = 0x41, 4, u8, wr
 @taint_intro 5, "file demo/input.txt", 4
 @context "mem32[0xdda0a004]" = 0xff, 5, u8, wr
 @taint_intro 6, "file demo/input.txt", 5
 @context "mem32[0xdda0a005]" = 0xff, 6, u8, wr
 @taint_intro 7, "file demo/input.txt", 6
 @context "mem32[0xdda0a006]" = 0x0, 7, u8, wr
 @taint_intro 8, "file demo/input.txt", 7
 @context "mem32[0xdda0a007]" = 0x0, 8, u8, wr
 @taint_intro 9, "file demo/input.txt", 8
 @context "mem32[0xdda0a008]" = 0x42, 9, u8, wr
```

Slicing with moflow

- On the next slide we will skip down to the first instruction executed after the read() system call returns.
- Before the first instruction description we see there are variable names given to each byte that was read in.
- Each time a memory location is written to it will get a new variable name assigned. This is called a Static Single Assignment form and simplifies our slicing
- These are in the form dsa_[“mem”|”REG”]_[address]_1_[unique ID]

```
dsa_mem_dda0affe_1_16372:u8 = symb_4095_8896:u8
dsa_mem_dda0afff_1_16373:u8 = symb_4096_8898:u8
addr 0xddae6318 @asm "movzx  eax,BYTE PTR [eax]" @tid "0"
  @context "R_EAX_32" = 0xdda0a000, 0, u32, wr
  @context "R_EAX_32" = 0xdda0a000, 0, u32, rd
  @context "mem32[0xdda0a000]" = 0x41, 1, u8, rd
label pc_0xddae6318
dsa_R_EAX_32_1_16374:u32 = 0xdda0a000:u32
dsa_R_EAX_32_1_16375:u32 = pad:u32(pad:u8(dsa_mem_dda0a000_1_12278:u8))
addr 0xddae5558 @asm "cmp    eax,0xffffffff" @tid "0"
  @context "R_EAX_32" = 0x41, 1, u32, rd
  @context "R_EFLAGS" = 0x286, 0, u32, wr
label pc_0xddae5558
dsa_R_ZF_1_16376:bool = false
dsa_R_AF_1_16377:bool = false
dsa_R_OF_1_16378:bool = false
dsa_R_SF_1_16379:bool = true
dsa_R_DF_1_16380:bool = false
dsa_R_CF_1_16381:bool = false
dsa_R_EFLAGS_1_16382:u32 = 0x286:u32
dsa_R_PF_1_16383:bool = true
```

Slicing with moflow

- That is a little cumbersome at first glance so let’s break it down

```
addr 0xddae6318 @asm "movzx  eax,BYTE PTR [eax]" @tid "0"
  @context "R_EAX_32" = 0xdda0a000, 0, u32, wr
  @context "R_EAX_32" = 0xdda0a000, 0, u32, rd
  @context "mem32[0xdda0a000]" = 0x41, 1, u8, rd
```

- This shows the values of EAX which was read, dereferenced, and written back in to EAX.
- The memory value in EAX that was dereferenced is shown on the last line. The field after the byte value is ‘1’ meaning it is the first byte from our input file

Slicing with moflow

- Lets quickly take a look at the exception that we were tracing in the debugger

```c
#define BUF_SIZE 1024
#define BIG 0x1000000

void read_file(char *fn, char *buffer, int size){
  FILE * pFile;
  size_t result;

  pFile = fopen (fn, "rb" );
  if (pFile==NULL) {fprintf(stderr, "Can't open %s", fn); exit (1);}

  result = fread (buffer,1,size,pFile);
  if (result != size)
    printf("read bytes %d < %d\n", result, size);

  fclose (pFile);
}
```

```c
struct header {
  char magic[4];
  int total_size; // read av on large total_size
  int record_count;
};

struct record {
  int type;      // read or subrecord
  int len;       // write av on long len
  char val[1];
};

void test_tlv_triage(char *buf)
{
  char *newbuf, *r_copy;
  int count, offset;
  struct header *h = (struct header *)buf;
  struct record *r = (struct record *)(buf + sizeof(struct header));
```

```c
if(memcmp(&h->magic, "AAAA", 4) != 0)
{
  printf("bad magic\n");
  return;
}
else
{
  printf("total_size: %d, record_count %d\n", h->total_size, h->record_count);
}
fflush(stdout);

newbuf = (char *)malloc(BUF_SIZE);
memcpy(newbuf, buf, h->total_size); // readAV if total_size > BUF_SIZE
```

```c
count = 0;
offset = sizeof(struct header);
while(count < h->record_count)
{
  printf("record: type %d, len %d\n", r->type, r->len);
  if(r->type == 1)
  {
    memcpy(newbuf, r->val, r->len);
  }
  else if(r->type == 2)
  {
    memcpy(newbuf + (int)&r->val, r->val + 4, r->len);
  }

  offset += r->len;
  count++;
}

return;
```

```
vulndev@vulndev-x64 /vulndev/demo $ xxd input.txt | head -2
0000000: 4141 4141 ffff 0000 4242 4242 0100 0000  AAAA....BBBB....
0000010: ffff 0000 0000 0000 0000 0000 0000 0000  ................
vulndev@vulndev-x64 /vulndev/demo $ ./demo tlv input.txt
total_size: 65535, record_count 1111638594
Segmentation fault
```

```
[──────────────────────────────────STACK──────────────────────────────────]
00:0000│ esp  0xffffcbc4 ◂— 0x0
01:0004│      0xffffcbc8 —▸ 0x55752000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1acda8
02:0008│      0xffffcbcc —▸ 0x80488f5 (test_tlv_triage+158) ◂— mov    dword ptr
[ebp - 0x1c], 0
03:000c│      0xffffcbd0 —▸ 0x804b008 ◂— 0x41414141 ('AAAA')
04:0010│      0xffffcbd4 —▸ 0xffffcc2c ◂— 0x41414141 ('AAAA')
05:0014│      0xffffcbd8 ◂— 0xffff
06:0018│      0xffffcbdc —▸ 0x80487e8 (read_file+155) —▸ 0x8955c3c9 ◂— 0x8955c3c
9
07:001c│      0xffffcbe0 —▸ 0x804b008 ◂— 0x41414141 ('AAAA')
[────────────────────────────────BACKTRACE────────────────────────────────]
 ► f 0 556daf06 __memcpy_ssse3_rep+1302
   f 1  80488f5 test_tlv_triage+158
   f 2  8048bb4 main+218
   f 3 555beaf3 __libc_start_main+243
Program received signal SIGSEGV (fault address 0xffffe000)
```

- We crashed in a memcpy. We know that if the user controls any of the arguments to a memcpy this could be a potentially exploitable bug

- We are not interested in the internals of libc, we can see that memcpy was called from our main image, so lets just get the tainted instructions in our main image

- One of these instructions required some sanity checking before the user controlled operands were passed as arguments to memcpy

```
root@d14e0cf39781:/moflow/slicer# grep "addr 0x804" /tmp/demo.trace.il
addr 0x80488a0 @asm "mov    edx,DWORD PTR [eax+0x8]" @tid "0"
addr 0x80488a6 @asm "mov    eax,DWORD PTR [eax+0x4]" @tid "0"
addr 0x80488a9 @asm "mov    DWORD PTR [esp+0x8],edx" @tid "0"
addr 0x80488ad @asm "mov    DWORD PTR [esp+0x4],eax" @tid "0"
addr 0x80488dc @asm "mov    eax,DWORD PTR [eax+0x4]" @tid "0"
addr 0x80488df @asm "mov    DWORD PTR [esp+0x8],eax" @tid "0"
addr 0x80488e3 @asm "mov    eax,DWORD PTR [ebp+0x8]" @tid "0"
addr 0x80488e6 @asm "mov    DWORD PTR [esp+0x4],eax" @tid "0"
```

- We could run the tool three times, slicing back on any byte from each of the three memcpy parameters. In the last instructions executed before calling memcpy we see that the instructions that were setting the “src” and “size” parameters to memcpy were tainted.

```
root@d14e0cf39781:/moflow/slicer# grep "addr 0x804" /tmp/demo.trace.il
addr 0x80488a0 @asm "mov    edx,DWORD PTR [eax+0x8]" @tid "0"
addr 0x80488a6 @asm "mov    eax,DWORD PTR [eax+0x4]" @tid "0"
addr 0x80488a9 @asm "mov    DWORD PTR [esp+0x8],edx" @tid "0"
addr 0x80488ad @asm "mov    DWORD PTR [esp+0x4],eax" @tid "0"
addr 0x80488dc @asm "mov    eax,DWORD PTR [eax+0x4]" @tid "0"
addr 0x80488df @asm "mov    DWORD PTR [esp+0x8],eax" @tid "0"
addr 0x80488e3 @asm "mov    eax,DWORD PTR [ebp+0x8]" @tid "0"
addr 0x80488e6 @asm "mov    DWORD PTR [esp+0x4],eax" @tid "0"
```

- EAX is being moved to the stack, so lets see the instruction before the mov to ESP+8 and grab the variable name of a memory byte that EAX was pointing to

```
root@d14e0cf39781:/moflow/slicer# tac /tmp/demo.trace.il | grep -B13 -m1 "esp+0x8"
addr 0x80488e3 @asm "mov    eax,DWORD PTR [ebp+0x8]" @tid "0"
dsa_mem_ffffd20b_1_24602:u8 = low:u8(dsa_R_EAX_32_1_24597:u32 >> 0x18:u32)
dsa_mem_ffffd20a_1_24601:u8 = low:u8(dsa_R_EAX_32_1_24597:u32 >> 0x10:u32)
dsa_mem_ffffd209_1_24600:u8 = low:u8(dsa_R_EAX_32_1_24597:u32 >> 8:u32)
dsa_mem_ffffd208_1_24599:u8 = low:u8(dsa_R_EAX_32_1_24597:u32)
dsa_R_ESP_32_1_24598:u32 = 0xffffd200:u32
label pc_0x80488df
  @context "mem32[0xffffd20b]" = 0x42, -1, u8, wr
  @context "mem32[0xffffd20a]" = 0x42, -1, u8, wr
  @context "mem32[0xffffd209]" = 0x42, -1, u8, wr
  @context "mem32[0xffffd208]" = 0x42, -1, u8, wr
  @context "R_EAX_32" = 0xffff, -1, u32, rd
  @context "R_ESP_32" = 0xffffd200, 0, u32, rd
addr 0x80488df @asm "mov    DWORD PTR [esp+0x8],eax" @tid "0"
```

- The dsa_mem variable names are the expected input for our slicer so we can pick one and now get a slice of only the instructions that touched that byte of data

```
root@d14e0cf39781:/moflow/slicer# ./slicer -il /tmp/demo.trace.il -var dsa_mem_ffffd20b
_1_24602 2>/dev/null | grep addr
addr 0xddaf3506 @asm "rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]"
addr 0x80488dc @asm "mov    eax,DWORD PTR [eax+0x4]" @tid "0"
addr 0x80488df @asm "mov    DWORD PTR [esp+0x8],eax" @tid "0"
```

- We see the path from libc copying the file from fread into a buffer and then copying values from that buffer to the length parameter to memcpy()

- Now we examine the operands to that first instruction to see the byte offsets in the file. They are shown below as byte 5, 6, 7, and 8 which are base 1. We can see the value was 65535 which was too large and undhecked , resulting in a ReadAV

```
root@d14e0cf39781:/moflow/slicer# ./slicer -il /tmp/demo.trace.il -var
dsa_mem_ffffd20b_1_24602 2>/dev/null | grep -A10 0xddaf3506
addr 0xddaf3506 @asm "rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]"
  @tid "0" @context "R_EDI_32" = 0xffffd260, 0, u32, rd
  @context "R_ESI_32" = 0xdda0a004, 0, u32, rd
  @context "R_ECX_32" = 0xff, -1, u32, rw
  @context "R_EFLAGS" = 0x246, 0, u32, rd
  @context "mem32[0xdda0a004]" = 0xff, 5, u8, rd
  @context "mem32[0xdda0a005]" = 0xff, 6, u8, rd
  @context "mem32[0xdda0a006]" = 0x0, 7, u8, rd
  @context "mem32[0xdda0a007]" = 0x0, 8, u8, rd
```

Thank you!

rjohnson@fuzzing.io

https://moflow.org

https://github.com/moflow