Google

Last Year in DFSan

Andrew Browne
2021-09-14

What is DFSan?

- DataFlow Sanitizer is a dynamic Taint Analysis tool
- Consists of Clang Instrumentation Pass and Run Time library
- Tracks tainted bytes through an execution
- Useful for answering:
  - Where does sensitive data flow to?
  - What code operates on tainted data (e.g. attacker controlled input)?
- DFSan API:
  - Set DFSan taint label on some bytes
  - Get DFSan taint label on some bytes

Overview

2020-09-01 ⟶ 2021-09-01

- Fast8 taint tracking data structure
- Origin Tracking
- Memory Allocation Improvements
- False Positive Dataflow Fixes
- Miscellaneous Improvements
- Future Work

Acknowledgement

Most of the work in the last year that I am presenting was done by others:

- Jianzhou Zhao
- George Balatsouras
- Matt Morehouse

Building on earlier DFSan and leveraging all the other sanitizer work.

Fast8 Mode

Legacy Mode vs Fast8 Mode

Legacy Mode

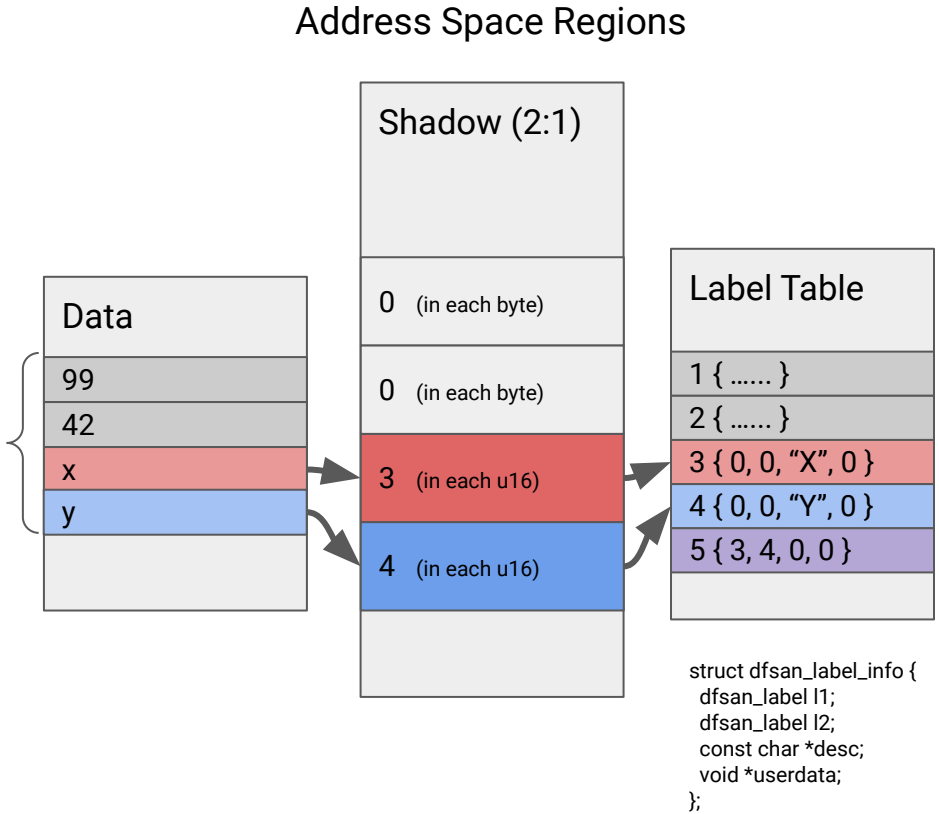
- 2^{16}-1 labels
- 2 byte Shadow per 1 byte App
- Shadow 16bit = table index
- New table entry for combination
- Deprecated & Deleted

Fast8 Mode

- 8 labels
- 1 byte Shadow per 1 byte App
- Shadow 8bit = label bitflags
- Bitwise OR to combine
- Need to run multiple times to track more than 8 things
- More similar to MSan

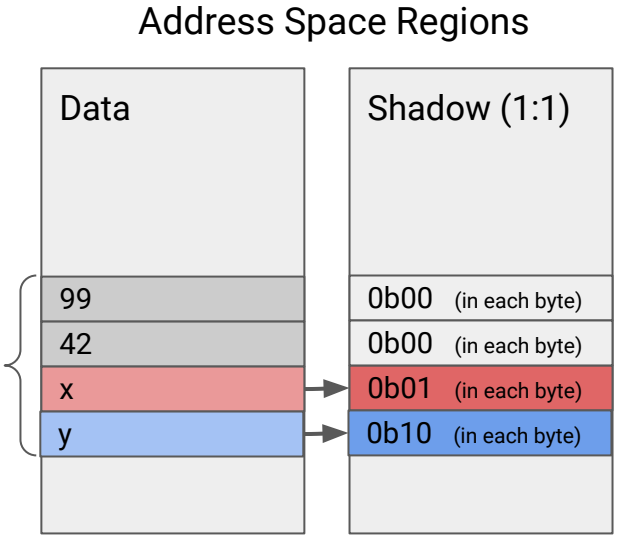
Legacy Mode - Illustration

```
int Example() {
   int x = get_input1();  // register has a shadow register
   dfsan_label labelC = dfsan_create_label("X", nullptr);
   dfsan_set_label(labelC, &x, sizeof(x)); // x label is 3

   int y = get_input2();
   dfsan_label labelD = dfsan_create_label("Y", nullptr);
   dfsan_set_label(labelD, &y, sizeof(y)); // y label is 4

   std::vector<int> things { 99, 42, x, y };

   int r = things[2] + things[3]; // label 5 = union(3,4)

   printf("%d\n",dfsan_get_label(r));  // prints: 5
   return r;   // label 5
}
```



Fast8 Mode - Illustration

```
int Example() {
   int x = get_input1();
   dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b00000001

   int y = get_input2();
   dfsan_set_label(1<<1, &y, sizeof(y)); // set label 0b00000010

   std::vector<int> things { 99, 42, x, y };

   int r = things[2] + things[3];   // label 0b11 = 0b01 | 0b10

   dfsan_get_label(r);  // label 0b00000011
   return r;   // label 0b00000011
}
```



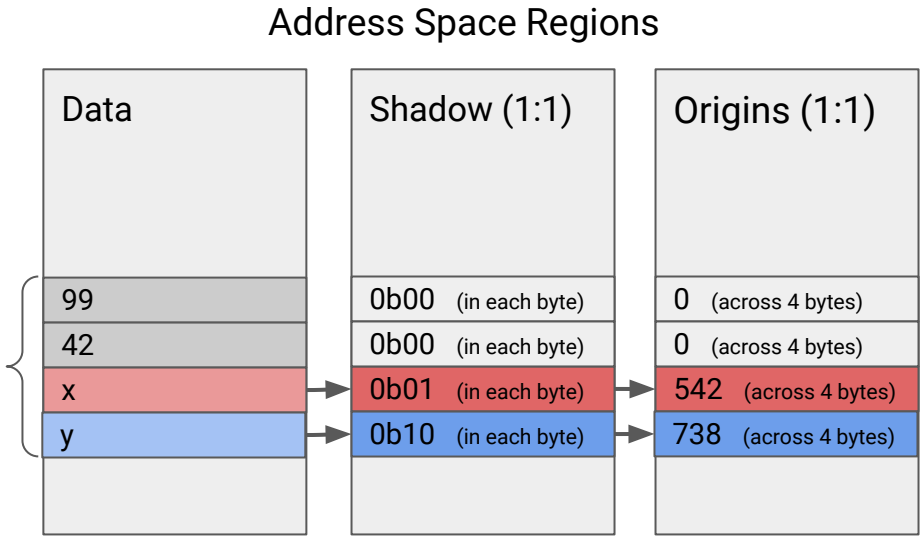
Origin Tracking

Origin Tracking

- DFSan says byte is tainted with label X… but **how**? Origin Tracking can help
- 1 extra byte Origins per 1 byte App
- Implementation based on MSan's Origin Tracking
- Record coarse information about the taint's value flow
  - dfsan_set_label → load → store → load → store → dfsan_get_label
- Each link in the chain has a capture of the current stack
- Only does Origin Tracking for >4 bytes of contiguous same-tainted data

Origin Tracking - Illustration

```
int Example() {
  int x = get_input1();
  dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b01

  int y = get_input2();
  dfsan_set_label(1<<1, &y, sizeof(y)); // set label 0b10

  std::vector<int> things { 99, 42, x, y };

  int r = things[2] + things[3];   // label 0b11 = 0b01 | 0b10

  dfsan_get_label(r);  // label 0b11
  return r;   // label 0b11
}
```



Origin Tracking - Illustration

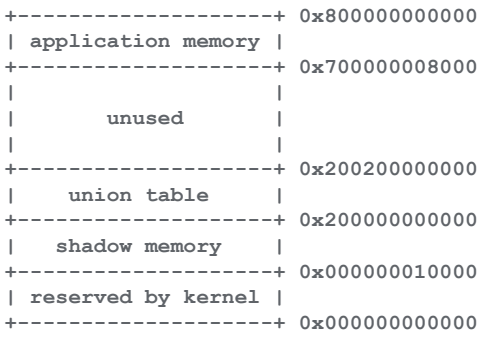
```
int Example() {
  int x = get_input1();
  dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b01

  int y = get_input2();
  dfsan_set_label(1<<1, &y, sizeof(y)); // set label 0b10

  std::vector<int> things { 99, 42, x, y };

  int r = things[2] + things[3];   // label 0b11 = 0b01 | 0b10

  dfsan_get_label(r);  // label 0b11
  return r;   // label 0b11
}
```

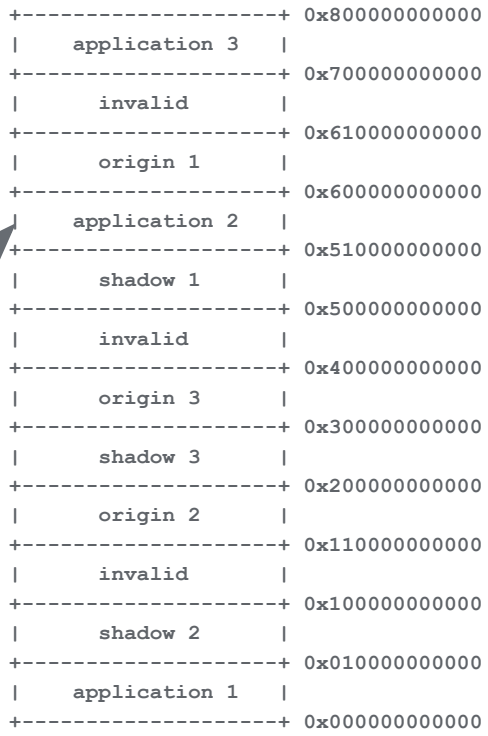
```
Taint value 0x3 (at 0x7ffcecc85b28) origin tracking ()
Origin value: 0x24400001, Taint value was stored to memory at
  #0 0x559bf7d8316a in Example() (.dfsan) /browneee/origins.cc:15:7
  #1 0x559bf7d83188 in main /browneee/origins.cc:36:11
  #2 0x7f1ecd4debbc in __libc_start_main (/.../libc.so.6+0x38bbc)
  #3 0x559bf7d53668 in _start /.../start.S:108

Origin value: 0x13400001, Taint value was stored to memory at
  #0 0x559bf7d83122 in __construct_range_forward<std::__u::allocator<
  #1 0x559bf7d83122 in __construct_at_end<const int *> /../include/c++
  #2 0x559bf7d83122 in vector /../include/c++/v1/vector:1357:9
  #3 0x559bf7d83122 in Example() (.dfsan) /browneee/origins.cc:13:20
  #4 0x559bf7d83188 in main /browneee/origins.cc:36:11
  #5 0x7f1ecd4debbc in __libc_start_main (/.../libc.so.6+0x38bbc)
  #6 0x559bf7d53668 in _start /.../start.S:108

Origin value: 0xfc00001, Taint value was created at
  #0 0x559bf7d82bcd in Example() (.dfsan) /browneee/origins.cc:8:3
  #1 0x559bf7d83188 in main /browneee/origins.cc:36:11
  #2 0x7f1ecd4debbc in __libc_start_main (/.../libc.so.6+0x38bbc)
  #3 0x559bf7d53668 in _start /.../start.S:108
```

Memory Allocation

Memory Layout now matches MSan

2020-09-01 → 2021-09-01





- Bug: Saw some Heap allocations here
- Some addresses shared shadow
- MSan mapping is more robust

Fixes for Releasing Memory

- Old implementation leaked a *LOT* of shadow memory
- Now uses Sanitizer Allocator
  - free() can also release Shadow and Origins
- Added munmap wrapper
  - can also release Shadow and Origins
- Writing 0 labels to shadow can release pages

False Positives Dataflow Fixes

Track subfields in composites separately

- Each field in a composite value should be tainted separately
- This fix is similar to MSan's behavior

```
std::pair<int, int> ReturnComposite() {
  int x = get_input1();
  dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b01
  return {42, x};
}

int y = ReturnComposite().first;    // FALSE POSITIVE: y has label 0b01  (FIXED)
```

Handle SelectInst Condition Correctly

- Label should not be propagated from SelectInst condition operand

```
int x = get_input1();
dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b01

int y = (x == 42) ? 100 : 200;  // FALSE POSITIVE: y has label 0b01  (FIXED)
```

Clear TLS after signal callback

- DFSan uses TLS to carry taint labels, particularly function arguments
- Signal handlers which touch TLS can set bits in TLS shadow

```
int x = get_input1();
dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b01
MyFunction(x);

int MyFunction(int x) {
  x;  // has label 0b01
  signal(SIGHUP, SIG_DFL); // trigger signal handler
  x;  // FALSE POSITIVE: x has label 0b101, should be label 0b01  (FIXED)
}
```

Known False Positive - std::vector<bool>

- DFSan labels apply to a whole byte
- std::vector<bool> packs 8 elements into one byte
  - conflates taint labels for all elements in the same byte
- Disable std::vector<bool> specialization?
  - Would change API, e.g. need std::vector<bool>::flip() in specialization
- Update libc++ std::vector<bool> specialization to maintain DFSan labels?
  - patch has too much `#ifdef DATAFLOW_SANITIZER`
- Discussion on https://reviews.llvm.org/D96842

Miscellaneous Improvements

Demangling

- Changed `dfs$` prefix → `.dfs` suffix
  - DFSan mangles all function names so that any external functions (not part of the compilation, and can't be instrumented) will cause a compilation error, instead of silently messing up the shadow data
  - This tells you when to add a DFSan wrapper or abi_list entry
  - Demangler supports `.dfs` suffix (as a vendor specific suffix)

Cleanups

- Removed dead code for unsupported platforms (only X86_64 Linux)
  - If someone wants to re-add support for other platforms, look to MSan
- Fixed Clang Tidy warnings
- Added various libc functions to ABI list
- Added more test cases
- Updated and improved documentation

Future Work

Control Flow Tainting

- Support for tracking implicit taint
- Two sides of the same coin:
  - What influence can tainted data have?
  - What information about sensitive data is leaked?

```
int x = get_input1();
dfsan_set_label(1<<0, &x, sizeof(x)); // set label 0b0001

int y = 100;
if (x == 42) {   // branch controlled by label 0b0001
  y = 200 + m; // has y implicit dependency on x (via control flow), label 0b0001 << 4 | label(m)
}                    //  e.g. only assign 4 labels, use 4 high bits to indicate control flow dependency
```

Continue Converging with MSan

- DFSan and MSan now operate in a very similar way
- More opportunities to share code

Conclusion

DFSan has grown more efficient and robust

- DFSan can now provide useful results on some of Google's largest C++ binaries
  - Runtime overheads are now reasonable
  - Overtainting has mostly been addressed