Not All Data are Created Equal:
Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks

Salman Ahmed¹, Hans Liljestrand², Hani Jamjoom¹, Matthew Hicks³, N. Asokan², Danfeng (Daphne) Yao³

¹ IBM Research
² University of Waterloo
³ Virginia Tech
Control-oriented attacks have become unreliable!

Control-oriented attacks
- Code injections
- Ret2libc
- ROP
- JOP
- COP
- COOP
- AOCR
- ...

Defenses
- Stack canaries
- ASLR
- Code diversification
- CPI
- Control-Flow Integrity (CFI)
- ...
- And many latest work (MLTA, TyPM) to make CFI sound and practical

Thus, recently we have seen an uptick to data-oriented attacks (e.g., DOP, BOP, ...
Why is the shift?

• **No violation** of the normal flow of a program (i.e., CFI won’t work)

• Expressiveness (DOP)

• Apparently, no practical defense mechanisms
Why are the existing defenses impractical?

Manipulation of data object/pointer is key attack strategy for data-oriented attacks.

It takes high overhead for data integrity!

- Data-Flow Integrity (DFI),
- Data-Space Randomization (DSR) and
- memory tagging techniques \{ 42% to 116% \}

- ARM Pointer Authentication \{ 19% - 26% \}

Due to huge number of data objects/pointers, on average \(~100x\) compared to code pointers!
Data Pointer Prioritization (DPP)

Fact:
It takes high overhead for existing defenses to prevent data-oriented attacks through data integrity!

Observation:
Not all data objects or their pointers are vulnerable or equally sensitive.

Idea:
We can filter out the insensitive data/pointers and protect only the sensitive ones through prioritization.

Goal: A framework that automatically identifies and prioritizes sensitive data objects/pointers.
- generic,
- adaptable, and
- capable of being integrated with existing defenses\(^1\).

\(^1\)ARM pointer authentication, Intel MPX, Hardware-assisted memory tagging, etc.
DPP Framework

External Input Sources

Data-Flow Analysis

Prioritization
(by applying rules)

Sensitive data objects/pointers

Vulnerability driven dataset

Rule-based heuristics

(taint identification & propagation)

Existing Defenses

ASan

PAuth

MPX

...

Softbound

CETS

External Input Sources

(network, filesystem, keyboard)
Challenges

• How to obtain representative set of rules with comprehensive coverage?
  ▪ Breaking down advanced exploits
    • generic rules, common components increase coverage

• How to evaluate the accuracy of the rules?
  ▪ Manually constructed ground truths from existing datasets
    • Juliet Test Suite, Linux Flaw Project, and data-oriented exploits
# DPP Rules

We extracted 7 rules in four categories.

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Category</th>
<th>Short Description</th>
<th>Example CVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>Control alteration</td>
<td>Data objects/pointers in predicates may alter program behavior</td>
<td>CVE-2006-5815</td>
</tr>
<tr>
<td>Rule 2</td>
<td>Control alteration</td>
<td>Data pointers used in loops may alter program flow or leak sensitive information</td>
<td>CVE-2006-5815</td>
</tr>
<tr>
<td>Rule 3</td>
<td>Proximity- based</td>
<td>Data pointers that are near to data buffers</td>
<td>CVE-2002-1496</td>
</tr>
<tr>
<td>Rule 4</td>
<td>Proximity- based</td>
<td>Data objects or pointers used in vulnerable functions</td>
<td>CVE-2021-31226</td>
</tr>
<tr>
<td>Rule 5</td>
<td>Erroneous</td>
<td>Data pointers that have been cast to different types</td>
<td>CVE-2018-6151</td>
</tr>
<tr>
<td>Rule 6</td>
<td>Erroneous</td>
<td>Data objects that have out-of-bound access</td>
<td>CVE-2021-21773</td>
</tr>
<tr>
<td>Rule 7</td>
<td>Unguarded</td>
<td>Pointers that have unbounded allocations</td>
<td>CVE-2020-11612</td>
</tr>
</tbody>
</table>
Construction of Data-Flow Graph

- We use SVF tool\(^1\) to construct the data/value flow graph for a program.

- SVF constructs a static data/value flow graph (SVFG) on top of LLVM IR.

- We addressed three missing dependencies in SVFG.

\(^1\)https://github.com/SVF-tools/SVF
#define MAX_SIZE 11
int global_array[100] = { -1 };

void testcase(int ts) {
    char buf[10];
    /if (ts > MAX_SIZE) exit(0);
    char *s = (char *) malloc(ts * sizeof(char));
    for (int i = 0; i < ts; i++) s[i] = i + '0';
    if (s[0] == 'a') printf("%s\n", s[ts-1]);
    memcpy(buf, s, strlen(s)); // sink
    printf("%s\n", buf);
}

int heap(int argc) {
    int array = (int *) malloc(sizeof(int) * 100);
    int res = array[argc + 100]; // overread
    free(array);
    return res;
}

int main(int argc, char **argv) {
    int size;
    scanf("%d", &size); // size' marked as tainted
    testcase(size);
    int r = heap(argc);
    printf("%d", r);
    return global_array[argc + 100]; // overread
}

Tainted Static Value Flow Graph (SVFG)
Implementation of DPP Rules

- Each rule is an LLVM analysis pass (LLVM 12)
- Rule 1, 2, 4, and 5: alias analysis, loop analysis, data layout information

Rule 3
data pointers follow data buffers

- Checking all local variables (alloca IR) in function
- Checking all global variables
- Checking any buffers followed by a pointer

Rule 6
Out-of-bound read/write

- Objects/pointers in Load, Store, and Call nodes
- Stack safety
- Size and offset approximation
Implementation of Rule 7

Dynamic memory allocation SVF nodes

Backward search in multiple paths for the mapped ICFG node for `cmp` instructions

Path explosion
Solution: Parameterized

Dealing with loops
Solution: removed loops

Relevant cmp instructions
Solution: common ancestor

Details in the paper
Evaluation

1) How capable and effective is DPP for prioritizing and ranking security critical data?

2) How much performance improvement can DPP enable?

Utilized Address Sanitizer (ASan) for the evaluation

Setup
- **baseline**: no instrumentation
- **asan**: instrumented all data objects
- **asan+dpp**: instrumented only prioritized data objects
Security Evaluation

ASan with DPP can detect all the memory errors Linux Flaw Project and Juliet Test Suite datasets the same as the default ASan can.

<table>
<thead>
<tr>
<th>CVE</th>
<th>Type</th>
<th>Application</th>
<th>ASan (default)</th>
<th>ASan + DPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2006-0539</td>
<td>heap-buffer-overflow</td>
<td>fcron-3.0.0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2006-2362</td>
<td>buffer-overflow</td>
<td>binutils-2.15</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-1759</td>
<td>stack-overflow</td>
<td>ctorrent-dh3.3.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2009-2285</td>
<td>heap-buffer-overflow</td>
<td>tiff-3.8.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2010-2481</td>
<td>out-of-order</td>
<td>tiff-3.9.2</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2010-2482</td>
<td>null-pointer-dereference</td>
<td>tiff-3.9.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2013-4243</td>
<td>heap-buffer-overflow</td>
<td>tiff-4.0.1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2013-4473</td>
<td>stack-smashing</td>
<td>poppler-0.24.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2013-4474</td>
<td>stack-buffer-overflow</td>
<td>poppler-0.24.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2014-1912</td>
<td>heap-buffer-overflow</td>
<td>Python-3.1.5</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2015-8668</td>
<td>heap-buffer-overflow</td>
<td>tiff-4.0.1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-10095</td>
<td>stack-buffer-overflow</td>
<td>tiff-4.0.7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-10271</td>
<td>heap-buffer-overflow</td>
<td>tiff-4.0.7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-12858</td>
<td>heap-use-after-free</td>
<td>libzip-1.2.0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-9138</td>
<td>stack-overflow</td>
<td>binutils-2.2.9</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Total tested cases</th>
<th>ASan (default)</th>
<th>ASan + DPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE121_Stack_Based_Buffer_Overflow</td>
<td>144</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>CWE122_Heap_Based_Buffer_Overflow</td>
<td>144</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>CWE124_Buffer_Underwrite</td>
<td>144</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>CWE126_Buffer_Overread</td>
<td>144</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>CWE127_Buffer_Underread</td>
<td>144</td>
<td>144</td>
<td>144</td>
</tr>
</tbody>
</table>

Juliet Test Suite

Linux Flaw Project
Prioritization Efficacy

More than 95% of data objects in a real-world program do not need protection.

- 16 vulnerable data objects from 13 applications

- DPP identifies potentially sensitive data objects by prioritizing top 3–4% data objects from real-world applications.
Performance Evaluation

DPP improves performance by $\sim 1.6x$
- Using SPEC CPU 2017 integer benchmark

DPP reduces run-time overhead by 70% compared to ASan.
- Using SPEC CPU 2017 integer benchmark
Limitations and Discussions

• Our current prototype is NOT a live defense.

• Our approach may miss sensitive objects if we overlook sensitive variables (apart from pointers)

• A broader benchmark is needed to fully assess the effectiveness of our rules.
Conclusion

• We proposed an automatic prioritization framework for identifying and ranking sensitive memory-resident data to prevent data-oriented attacks.

• Simple rule-based heuristics are effective.

• Our proposed prioritization scheme is new and different from the conventional protection paradigm.
Not All Data are Created Equal:
Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks

Thank You!

Q & A

https://github.com/salmanyam/DPP

Salman Ahmed

sahmed@ibm.com
@salmanyam
@salmanyam