Identifying Valuable Pointers In Heap Data

James Roney, Troy Appel, Prateek Pinisetti, James Mickens
Overview

• Data-oriented attacks manipulate programs while respecting control flow integrity

• Memory cartography is a powerful data-oriented attack
  • An attacker builds a map of pointers between memory regions (e.g., stack, heap, static data)
  • A memory read vulnerability in one region allows the attacker to navigate between regions and read data from the entire address space....
  • ... assuming that pointers reside constant offsets within regions!

• Stack and heap regions often have nondeterministic pointer offsets
• We show that an attacker with a memory read vulnerability can identify pointers using a signature-matching algorithm, even in nondeterministic regions
Outline

• Data-Oriented Attacks
• Memory Cartography
• Finding Pointers on The Heap
• Experiments
• Conclusion
Outline

• Data-Oriented Attacks
  • Memory Cartography
  • Finding Pointers on The Heap
• Experiments
• Conclusion
Data-Oriented Attacks

• Historically, attackers used memory bugs to subvert control flows
Data-Oriented Attacks

- However, modern mitigations make this more difficult

```c
x = do_authentication(user);

int authenticated = 0;
char password[128];
read_packet(password);
if(auth_pw(user, password){...}

authenticated = 1;
return authenticated;
```

```c
x = do_authentication(user);

int authenticated = 0;
char password[128];
read_packet(password);
if(auth_pw(user, password){...}

authenticated = 1;
return authenticated;
```

CFG Violation can be detected/prevented by ASLR, stack canaries, DEP, shadow stacks, etc.
Data-Oriented Attacks

- Data-oriented exploits avoid modifying control data, respecting the CFG.
Outline

• Data-Oriented Attacks
• Memory Cartography
• Finding Pointers on The Heap
• Experiments
• Conclusion
Memory Cartography

• Data-Oriented exploit introduced by Rogowski et al. (2018)
  • Attacker has a local read vulnerability
  • Wants to read from the entire address space without triggering a segmentation fault
  • Difficult due to fragmented nature of memory allocations

• Assumptions:
  • ASLR, DEP, stack canaries, etc. are enabled
  • Attacker can run victim binary locally
Memory Cartography

• Web Browser Example

Read Vulnerability Here

Want to read Cookies here
Memory Cartography

- Attacker runs binary locally, scans static data sections for inter-region pointers
- Records pointers in form $(\langle \text{src\_name}, \text{src\_offset} \rangle, \langle \text{dst\_name}, \text{dst\_offset} \rangle)$
- ASLR preserves relative offsets, so these tuples will be consistent across program runs when src and dst are static data regions
Memory Cartography

- Some of these pointers may simply be pointer-sized regions that happen to reference external memory regions.
- To filter out “false pointers,” the attacker repeats the procedure for multiple independent program loads, and looks for pointers that are consistently present.
Memory Cartography

- Process results in the ability to navigate across data sections and reach target heap.
- Still need a way of jumping from JS heap to a data section. Offsets of pointers in JS heap may not be consistent!
Rogowski et al. accomplished this with a heap spray of easily-recognizable objects containing known data section pointers. However, this approach may not be viable for all applications.
Outline

• Data-Oriented Attacks
• Memory Cartography
• Finding Pointers on The Heap
• Experiments
• Conclusion
Finding Pointers on the Heap

• Like in original cartography setup, attacker has a local heap read vulnerability, wants to read from the entire address space without triggering a fault

• However, the attacker has no influence over contents of the heap. So to find a pointer to another region, the attacker must scan the heap at attack-time and recognize the pointer somehow

• Assumptions:
  • ASLR, DEP, stack canaries, shadow stacks, etc. are enabled
  • Attacker can run the program locally
Finding Pointers on the Heap

• High-level idea:
  • Run the program locally several times, and identify recurring pointers to specific offsets within data sections
  • Use the bytes surrounding those frequent pointers to build an identifiable “signature”
  • At attack-time, scan the heap using a local read vulnerability and match bytes to the signature from offline analysis
  • If the bytes surrounding an aligned, pointer-sized region match the signature, follow the pointer-sized region to a known offset within a data section
  • From there, perform further memory cartography as normal
Finding Pointers on the Heap

- Attacker runs the program locally and determines the boundaries of allocated regions (by looking at /proc/<pid>/maps, for example)
- Note that the “heap” can actually comprise multiple VMAs (as when the program uses an mmap-based allocator)
Finding Pointers on the Heap

• Attacker then scans the heap, looking for pointers to other regions, and identifies the most frequent pointer destinations
  • “Most frequent” meaning the \((\text{dst} \_\text{name}, \text{dst} \_\text{offset})\) pairs that were observed the most times across multiple program runs
Finding Pointers on the Heap

• Attacker examines the bytes surrounding pointers to frequent destinations
Finding Pointers on the Heap

- Attacker uses bytes surrounding pointers to build a filter
- Filter is simply a sequence of lower bounds and upper bounds on each byte in a fixed-width window surrounding the pointer. Filter bounds are determined by taking the highest and lowest byte value observed in each position during local program runs
Finding Pointers on the Heap

• Finally, filter bounds are used to identify a pointer to a known destination during an attack-time memory scan.
Finding Pointers on the Heap

• Finally, filter bounds are used to identify a pointer to a known destination during an attack-time memory scan
Finding Pointers on the Heap

• Finally, filter bounds are used to identify a pointer to a known destination during an attack-time memory scan.
Finding Pointers on the Heap

- Finally, filter bounds are used to identify a pointer to a known destination during an attack-time memory scan.
Finding Pointers on the Heap

• Finally, filter bounds are used to identify a pointer to a known destination during an attack-time memory scan.
Outline

• Data-Oriented Attacks
• Memory Cartography
• Finding Pointers on The Heap
• Experiments
• Conclusion
Methodology

• General setup: run the program 10 times, dumping memory each time
  • Each run is a fresh ASLR load

• Use the first nine program runs to compute filters as described previously
  • Create and evaluate pointers for the four most frequent pointer destinations observed across all runs

• Use the holdout run to test the accuracy of the filter in identifying the pointer of interest
  • We hold out each run one-by-one and average the results (10-fold cross validation)
Methodology

• To test the performance of a filter, we simply ran it over all aligned pointer-sized regions in the dumped heap from a held-out run
  • Future work should demonstrate an end-to-end attack with a real read vulnerability. We assumed the presence of such a vulnerability and simulated it by dumping the heap

• Filter performance metrics:

\[
\text{Precision} = \frac{\text{True Filter Matches}}{\text{True Filter Matches} + \text{False Filter Matches}}
\]

\[
\text{Recall} = \frac{\text{True Filter Matches}}{\text{Total True Pointers Scanned}}
\]
Experiments: Vim

- Simple single-threaded test program with a single, well-defined heap region

<table>
<thead>
<tr>
<th>Rank</th>
<th>Region</th>
<th>Offset</th>
<th>True Positives</th>
<th>False Positives</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>vim_basic_4</td>
<td>90912</td>
<td>25650 / 25650</td>
<td>0 / 1525680</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>libc-2.31.so_5</td>
<td>3040</td>
<td>25451 / 25452</td>
<td>160 / 3077198</td>
<td>.994</td>
<td>.999</td>
</tr>
<tr>
<td>12</td>
<td>libc-2.31.so_5</td>
<td>2816</td>
<td>2360 / 2361</td>
<td>1375 / 3100289</td>
<td>.632</td>
<td>.999</td>
</tr>
<tr>
<td>14</td>
<td>libc-2.31.so_5</td>
<td>3072</td>
<td>800 / 802</td>
<td>1378 / 3101848</td>
<td>.367</td>
<td>.998</td>
</tr>
</tbody>
</table>
Experiments: Firefox

- Wanted to simulate vulnerability in JS engine heap
- Unlike Vim, Firefox JS engine uses an mmap-based allocator (jemalloc), so the heap is spread over multiple VMAs
- Identified jemalloc heap “chunks” by size, treated the aggregate contents of these regions as the effective program heap

<table>
<thead>
<tr>
<th>Rank</th>
<th>Region</th>
<th>Offset</th>
<th>True Positives</th>
<th>False Positives</th>
<th>Precision</th>
<th>Recall</th>
<th>Precision (worst region)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>libxul.so_2</td>
<td>21438312</td>
<td>310724 / 310735</td>
<td>662 / 6242515</td>
<td>.998</td>
<td>.999</td>
<td>.988</td>
</tr>
<tr>
<td>2</td>
<td>libxul.so_2</td>
<td>21438264</td>
<td>299715 / 299716</td>
<td>755 / 6253534</td>
<td>.997</td>
<td>.999</td>
<td>.993</td>
</tr>
<tr>
<td>3</td>
<td>libxul.so_1</td>
<td>27080560</td>
<td>23704 / 25603</td>
<td>2369 / 1612697</td>
<td>.909</td>
<td>.926</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>libxul.so_1</td>
<td>27085200</td>
<td>17300 / 18850</td>
<td>38 / 800250</td>
<td>.998</td>
<td>.918</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Experiments: Firefox

- Indicates the worst-case performance if attacker were limited to reading from a single randomly-chosen heap chunk

<table>
<thead>
<tr>
<th>Rank</th>
<th>Region</th>
<th>Offset</th>
<th>True Positives</th>
<th>False Positives</th>
<th>Precision</th>
<th>Recall</th>
<th>Precision (worst region)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>libxul.so_2</td>
<td>21438312</td>
<td>310724 / 310735</td>
<td>662 / 6242515</td>
<td>.998</td>
<td>.999</td>
<td>.988</td>
</tr>
<tr>
<td>2</td>
<td>libxul.so_2</td>
<td>21438264</td>
<td>299715 / 299716</td>
<td>755 / 6253534</td>
<td>.997</td>
<td>.999</td>
<td>.993</td>
</tr>
<tr>
<td>3</td>
<td>libxul.so_1</td>
<td>27080560</td>
<td>23704 / 25603</td>
<td>2369 / 1612697</td>
<td>.909</td>
<td>.926</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>libxul.so_1</td>
<td>27085200</td>
<td>17300 / 18850</td>
<td>38 / 800250</td>
<td>.998</td>
<td>.918</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Experiments: Apache

• Used OpenSSL 1.0.1, which is vulnerable to HeartBleed
• Identified pointers in heap region containing the vulnerable HeartBleed buffer
• Served a WordPress site with simulated traffic
Experiments: Take-home Point

• In all tested programs, we were able to identify pointers to static data sections with very high precision

• We were able to reliably reach static data sections with high connectivity to the rest of the address space, making them ideal starting points for memory cartography attacks

• This means powerful memory cartography attacks are possible even when the attacker has no control of the heap layout
Outline

• Data-Oriented Attacks
• Memory Cartography
• Finding Pointers on The Heap
• Experiments
• Conclusion
Conclusions

• A simple signature-matching algorithm facilitates powerful memory cartography attacks, even when the attacker does not have control over heap contents

• Some caveats:
  • As in original memory cartography paper, assumes that inter-region pointers are located at the same offsets on the local machine and the victim machine
  • Time/bandwidth constraints imposed by real-world exploits may limit the attacker’s ability to scan the entire heap