Dynamic Data Structure Excavation
or “Gimme back my symbol table!”

Asia Slowinska, Traian Stancescu,
Herbert Bos
VU University Amsterdam
Anonymous bytes only...
Goals

- Long term: reverse engineer complex software
Goals

• Long term: reverse engineer complex software
• Short term: reverse engineer data structures

```
push    %ebp
mov     %esp,%ebp
sub     $0x8,%esp
mov     0x8(%ebp),%eax
lea     -0x8(%ebp),%ecx
mov     %eax,%edx
mov     $0x8c,%eax
mov     %eax,0x8(%esp)
mov     %edx,0x4(%esp)
mov     %ecx,(%esp)
call    0x29
mov     0x8(%ebp),%eax
leave
ret
nop
nop
```

```
struct s1{
    char f1 [128];
    int f2;
    int f3;
    int f4;
};
struct s1*
fun1 (struct s1* a1)
{
    struct s1 l1;
}
```
WHY?
Application I: legacy binary protection

• Legacy binaries everywhere
• We suspect they are vulnerable

But...

How to protect legacy code from memory corruption?
Answer: find the buffers and make sure that all accesses to them do not stray beyond array bounds.
Application II: binary analysis

• We found a suspicious binary – is it malware?
• A program crashed… - let’s investigate!

But…

Without symbols, what can we do?
Answer: generate the symbols ourselves!
(demo later)
Why is it difficult?

1. `struct employee` {
   2. `char name[128];`
   3. `int year;`
   4. `int month;`
   5. `int day;`
   6. `}

8. `struct employee e;`
9. `e.year = 2010;`

MISSING
- Data structures
- Semantics
1. struct employee {
2.   char name[128];
3.   int year;
4.   int month;
5.   int day
6. };
7.
8. struct employee e;
9. e.year = 2010;

Yes, data is unstructured...
But – usage is NOT!
Data structures: key insight

Yes, data is unstructured...
But – usage is NOT!
Data structures: key insight

```
1. struct employee {
2.     char name[128];
3.     int year;
4.     int month;
5.     int day
6. };
7. 
8. struct employee e;
9. e.year = 2010;
```

Analyse dynamically

KLEE/S²E

inputs

Emulator

app

data structures
Intuition

- Observe how memory is *used* at runtime to detect data structures
  - E.g., if A is a pointer...

1. and A is a function frame pointer, then *(A + 8) is perhaps a function argument

2. and A is an address of a structure, then *(A + 8) is perhaps a field in this structure

3. and A is an address of an array, then *(A + 8) is perhaps an element of this array
Arrays are tricky

Access pattern & detection:

• `elem = next++;`
  
  – Look for chains of accesses in a loop
Arrays are tricky

Access pattern & detection:

• `elem = next++;`
  – Look for chains of accesses in a loop

• `elem = array[i];`
  – Look for sets of accesses with the same base in a linear space
Arrays are tricky

Access pattern & detection:

- \( \text{elem} = \text{next}++; \)
  - Look for chains of accesses in a loop
- \( \text{elem} = \text{array}[i]; \)
  - Look for sets of accesses with the same base in a linear space

Challenges:

- Boundary elements accessed outside the loop
- Nested loops
- Multiple loops in sequence
More challenges

Examples:

- Decide which memory accesses are relevant
  - Problems caused by e.g., \texttt{memset}-like functions

\begin{itemize}
\item \texttt{structure}
\item \texttt{array 1}
\item \texttt{array 2}
\end{itemize}

Suggested by \texttt{memset}
More challenges

Examples:
• Decide which memory accesses are relevant
  – Problems caused by e.g., memset-like functions
• Even more in the paper 😊
Results in terms of accuracy – heap memory

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
</tr>
</tbody>
</table>

Unused arrays
Flattened
Unused
Missed
Ok

Heap Memory

bytes
variables
demo now
Conclusions

• We *can* recover data structures by tracking memory accesses
• We believe we can protect legacy binaries
• We are working on data coverage