Sleak: Automating Address Space Layout Derandomization

Christophe Hauser¹, Jayakrishna Menon¹,² Yan Shoshitaishvili², Ruoyu Wang², Giovanni Vigna³, Christopher Kruegel³

Information Sciences Institute, University of Southern California¹
Arizona State University²
University of California, Santa Barbara³
Sleak in a Nutshell

What
- Sleak automates the process of discovering address leaks in binary programs.
- It detects **partial** and **indirect** leaks as well (i.e., leaking some bits of address).

How
- I performs static analysis and symbolic execution in order to generate precise expressions of what leaks.

Why
- I helps attackers bypass ASLR by recovering bits of leaked addresses.
Background: exploiting memory corruption bugs on modern OS platforms
The C programming language :)  

- Is almost 50 years old!
- In the top 10 programming languages used in 2019.
- The majority of our software stacks are still written in C (and C++).
- It compiles to binary and runs efficiently.
The C programming language :)  

- Is almost 50 years old!
- In the top 10 programming languages used in 2019.
- The majority of our software stacks are still written in C (and C++).
- It compiles to binary and runs efficiently.

Memory safety :(  

- Memory corruption bugs remain very common.
- It remains the most exploited class of bugs.

Memory corruption: memory modified with no assignments, e.g., buffer overflows, arbitrary writes
Fast forward: mitigations

Compiler-level mitigations
OS-level mitigations

1997: return to libc
2003: OpenBSD’s W^X
2004: DEP (MS Windows)
2005: ASLR in Linux
2007: ASLR in MacOS and Windows
2007: Return-oriented programming

Non-executable data
Fast forward: mitigations

Compiler-level mitigations
OS-level mitigations

2003: OpenBSD’s W^X
2005: ASLR in Linux
2007: Return-oriented programming

1997: return to libc
2004: DEP (MS Windows)
2007: ASLR in MacOS and Windows

Address Space Layout Randomization
Non-executable data pages

W^X: “Write xor Execute”
DEP: Data Execution Prevention

Classic BOF
Stack
RET
...
...
BUF
...
...
Libc
...
Main binary
Code

@BUF
Overflow
Non-executable data pages

W^X: “Write xor Execute”
DEP: Data Execution Prevention
**Non-executable data pages**

W^X: “Write xor Execute”
DEP: Data Execution Prevention

- **Attackers employ code reuse attacks.**
- W^X, DEP: non-executable data pages
  - Attackers employ code reuse attacks
- ASLR: Address Space Layout Randomization - the base address of .text, .data, heap, stack and memory mappings is randomized
  - Attackers need pointer leaks!
While (i < size) {
    b = buf[i];
    ...
}
While ($i < \text{size}$)
{
    $b = \text{buf}[i]$;
    ...
}

Attacker-controlled

*f;
Int x;
Char *buf;

OOB read!
Pointer leak example

Attacker-controlled

While (i < size) {
    b = buf[i];
    ...
}

0xAB200

*\textbf{f};

Int \textbf{x};

Char *buf;

OOB read!
Pointer leak example

While (i < size) {
    b = buf[i];
    ...
}

Attacker-controlled

Base address = 0xAB200 - 0x200

Base address = 0xAB200 - 0x200
ASLR weaknesses

- Leaking a single address is generally enough to recover the layout of an entire module (e.g., library).

- The entropy is limited by practical constraints (e.g., user/kernel separation, stack located higher than heap, etc.)
ASLR weaknesses

- Prior work has demonstrated that up to 20-bit of address-entropy remains within the reach of practical attacks.

- As a result partially leaking addresses can be sufficient for successful attacks.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Stack</th>
<th>Heap</th>
<th>Mmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>19</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>64-bit</td>
<td>30</td>
<td>28</td>
<td>28</td>
</tr>
</tbody>
</table>

*Bits of entropy per memory region (Linux 4.5.0)*
Enough background, now, Sleave!
Sleak in a Nutshell

- Consider a program with outputs $o_1, \ldots, o_n$ and addresses $a_1, \ldots, a_n$.

- We are interested in outputs leaking any transformation $f_k$ of an address, i.e.,:

$$\{ o_i = f_k (a_i) \}$$
Assumptions

- Stripped (Linux) binaries.
- Standard input/output implementations (i.e., we rule out custom input/output functions).
- Standard compiler, calling conventions...
Challenges
Source code

- Types.
- Variable names.
- Functions.
- ...

Binary

- Registers.
- Memory locations.
- Basic blocks.
- ...

Source code

- Types.
- Variable names.
- Functions.
- ...

Binary

- Registers.
- Memory locations.
- Basic blocks.
- ...
Static analysis

- Scalable.
- Imprecise.

Symbolic execution

- Precise.
- Unscalable.
Dynamic analysis

- Precise.
- Low coverage.
1. Path selection
2. Address identification
3. Leak identification
4. De-randomization
Static analysis: Path selection and address identification
1: Output function identification

- Control-Flow recovery.
- Identify statements corresponding to output function calls.
- Those are marked as *sinks*. 
2: Address identification

- Static backward slicing.
- Locate program statements defining addresses.
- Leverages address inference rules.
- Consider those as sources.
Address Inference Rules

(1) Leverage known information (GOT, relocations, external function prototypes, e.g., return values of malloc() or mmap()).

(2) Leverage instruction semantics (i.e., target of load and store operations)

(3) Value range: does the value fall within the .text, .data, heap, stack or memory mapping regions?
3: Leak Identification

- Paths between sources and sinks are symbolically executed.
- To limit state explosion, the execution is constrained to those paths defined statically.
- The symbolic expressions of output parameters are analyzed. Expressions depending on an address are flagged.
4. De-randomization

- An remote attacker observes the output $o_k$ of the program.
- The attacker obtains the expression of $o_k$ from Sleak.
- Using a constraint solver, the attacker guesses possible values for the leaked address.

$\text{Execution}$

$0_k = 42$

$\text{Binary}$

$0_k = a_4 / 4$

$\text{z3.solve(o== 42, o== a/4)}$

(a = 168, o = 42)
(a = 169, o = 42)
(a = 170, o = 42)
(a = 171, o = 42)
Evaluation
80 CTF binaries (Deconf quals 2012-2018)

OverlayFS (Linux Kernel)

libXSLT (large library used by Firefox and Chrome)

- angr (built-in analyses + custom module)
- Lightly modified Qemu
- Xeon E5-1650 v4 @ 3.60GHz CPUs and 64GB
Experimental setup

- We collected ground truth data from CTF writeups, manual analysis and the Common Vulnerabilities and Exposures (CVE) database.

- libXSLT and OverlayFS are complex code bases with extensive use of dynamic constructs. Therefore, we leverage dynamic execution to initialize the program state.
  - XSLT: test cases shipping with the library.
  - OverlayFS: benchmark of file system operations.
## Evaluation results (summary)

<table>
<thead>
<tr>
<th>CTF binaries</th>
<th>CFG Nodes</th>
<th>Functions</th>
<th>Sinks</th>
<th>Leak Detected</th>
<th>Ground Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00ctf_17_left</td>
<td>72</td>
<td>1</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>a5afefdb29d5dc067ed6507d78853c691</td>
<td>496</td>
<td>16</td>
<td>11</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>defcon_16_heapfun4u</td>
<td>200</td>
<td>5</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ez_pz</td>
<td>91</td>
<td>2</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>pwn1</td>
<td>318</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>int3errupted</td>
<td>327</td>
<td>6</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libXSLT</td>
<td>76842</td>
<td>505</td>
<td>27</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Overlayfs</td>
<td>1981</td>
<td>191</td>
<td>27</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Limitations

- Tracking data dependency on complex data structures on long code paths is hard!

- Static analysis / code coverage vs dynamic aspects of code (e.g. runtime binding).

- State explosion
  - Paths with complex loops.
  - Symbolic strings.

- Environment models (e.g., system calls).
Stumbling blocks

Data structure recovery.

Pointer aliasing.
Conclusion

- Sleak allows attackers to recover information about the memory layout applications in the presence of address space randomization.

- It is the first model to reason about indirect address leaks at the binary level.

- We evaluated it on both small userspace programs, a complex library and a kernel file system.