Sleak: Automating Address Space Layout Derandomization

Christophe Hauser[1] , Jayakrishna Menon[1],[2] Yan Shoshitaishvili[2] ,
Ruoyu Wang[2] , Giovanni Vigna[3] , Christopher Kruegel[3]

Information Sciences Institute, University of Southern California[1]
Arizona State University[2]
University of California, Santa Barbara[3]

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.

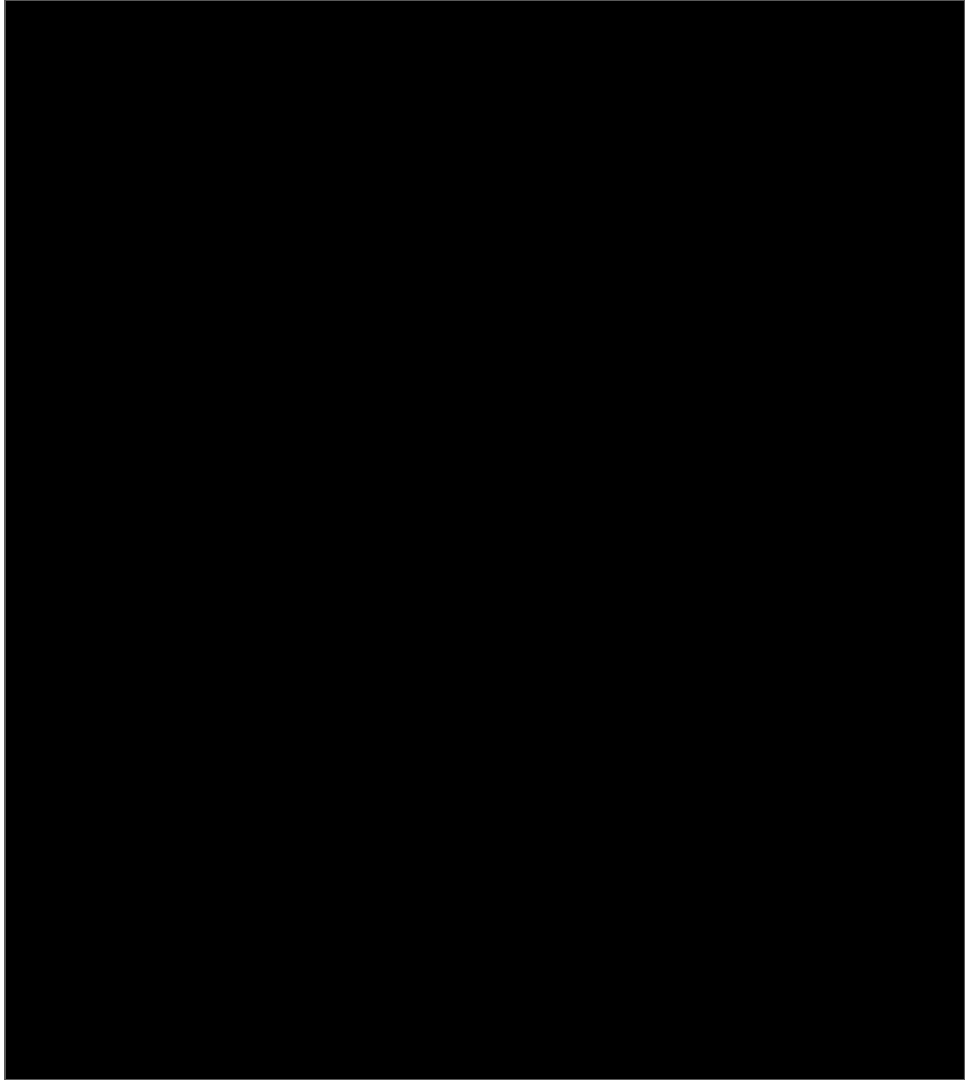
Why

- 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.

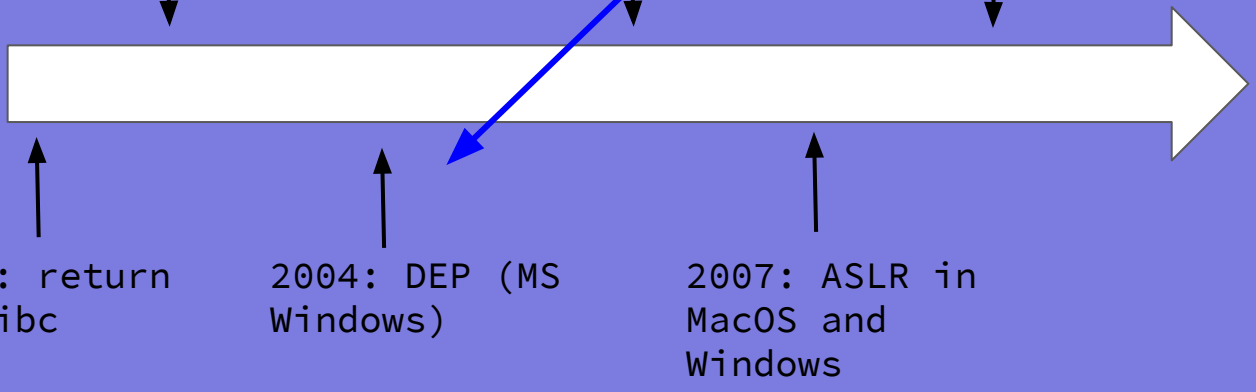
Memory corruption: memory modified with no assignments, e.g., buffer overflows, arbitrary writes

Fast forward: mitigations

Non-executable data

~~Compiler-level mitigations~~

OS-level mitigations

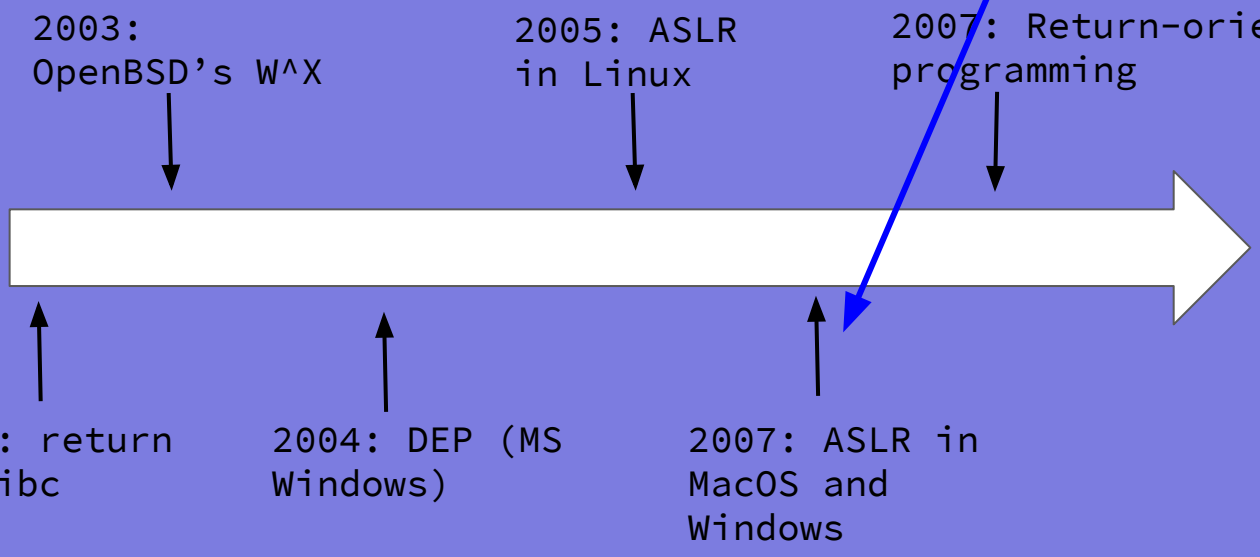


Fast forward: mitigations

~~Compiler-level mitigations~~

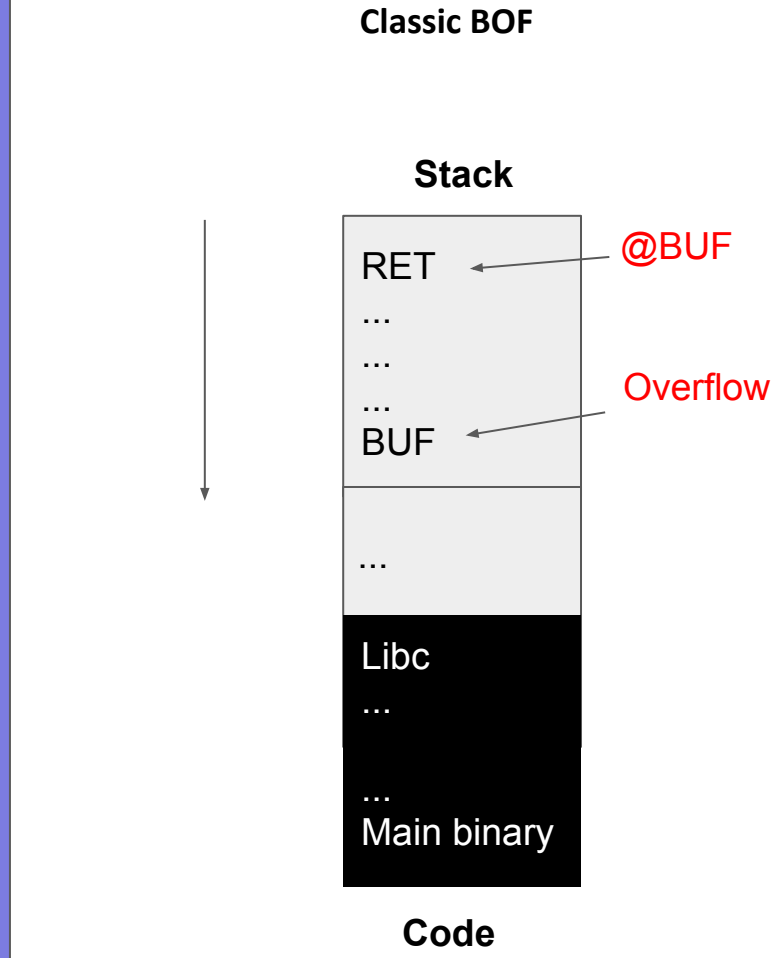
OS-level mitigations

Address Space Layout Randomization



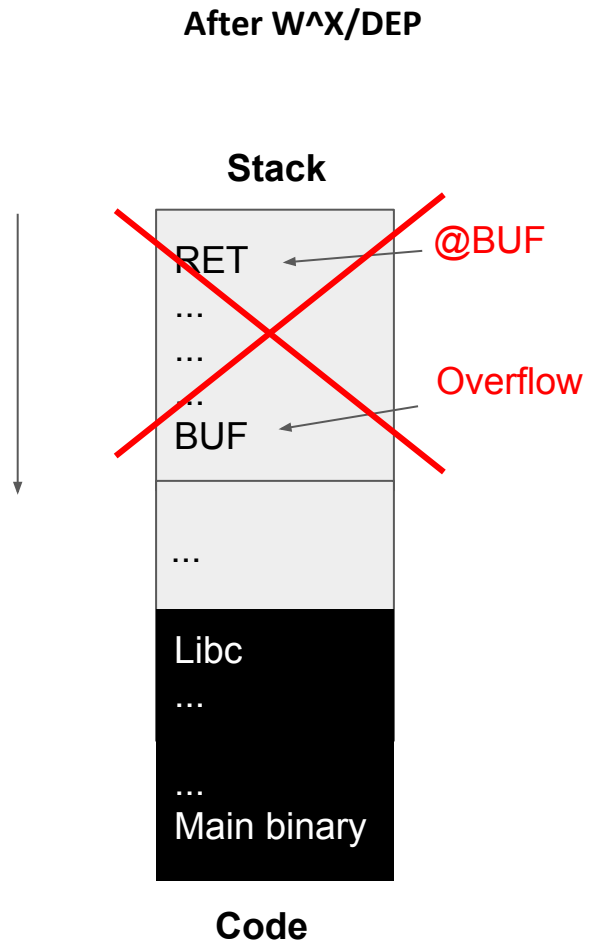
Non-executable data pages

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



Non-executable data pages

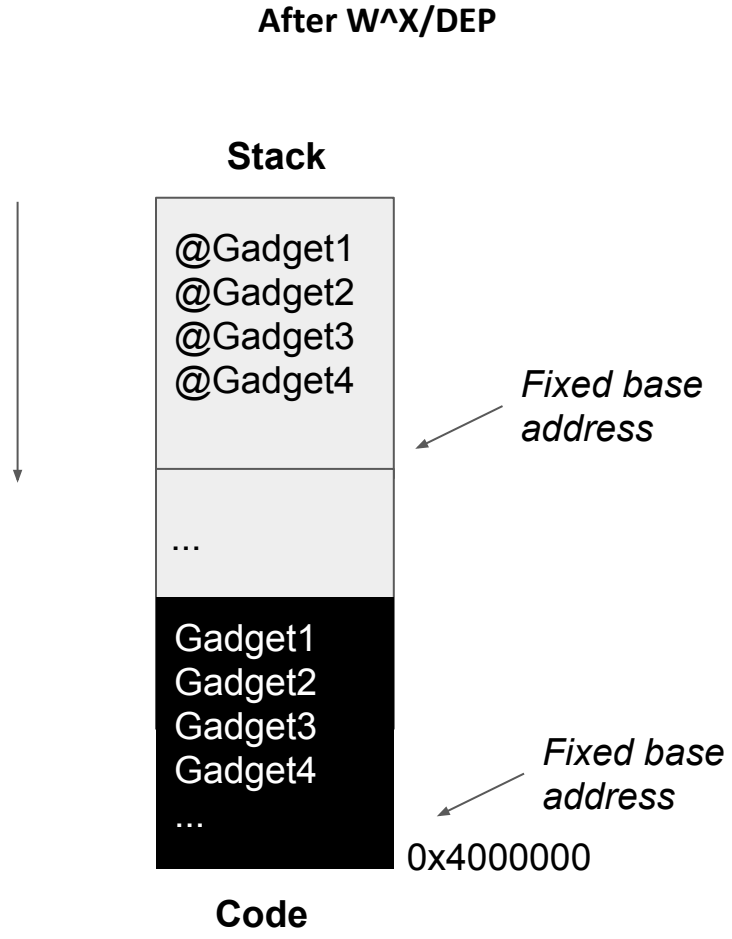
W^X: “Write xor Execute”
DEP: Data Execution Prevention



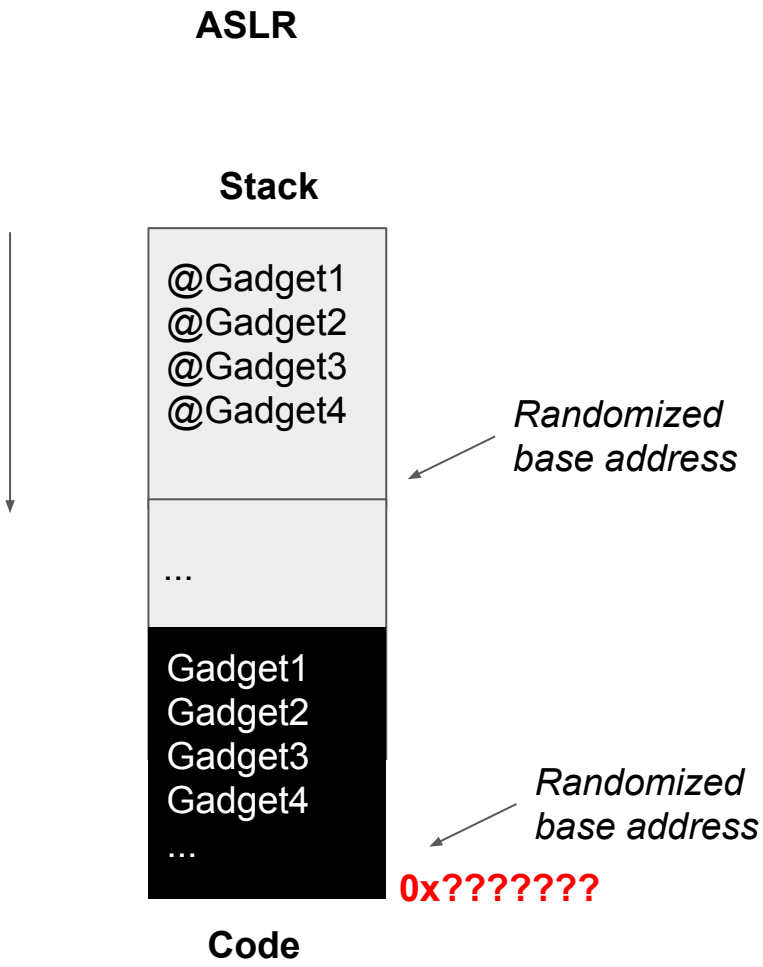
Non-executable data pages

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!***



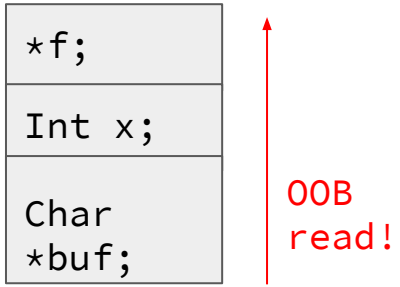
Pointer leak example

Attacker-controlled

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

Attacker-controlled

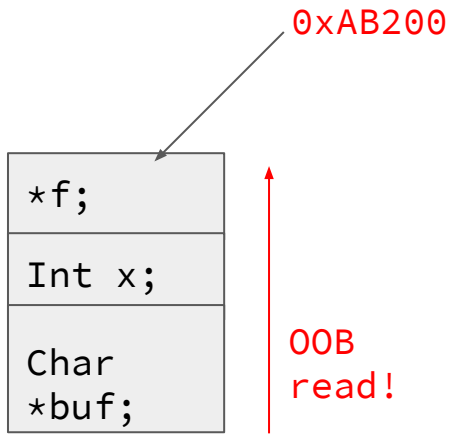
```
While (i < size)
{
     b = buf[i];
     …
}
```



Pointer leak example

Attacker-controlled

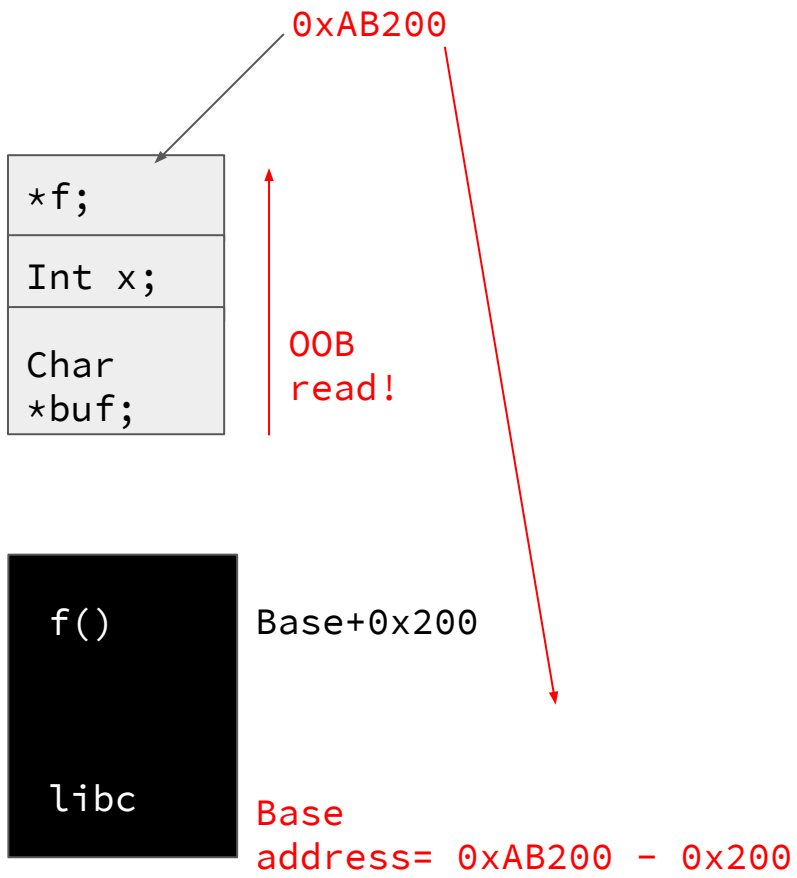
```
While (i < size)
{
     b = buf[i];
     …
}
```



Pointer leak example

Attacker-controlled

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



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.

Architecture	**Stack**	**Heap**	**Mmap**
32-bit	19	13	8
64-bit	30	28	28

Bits of entropy per memory region (Linux 4.5.0)

Enough background, now, Sleak!

Sleak in a Nutshell

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

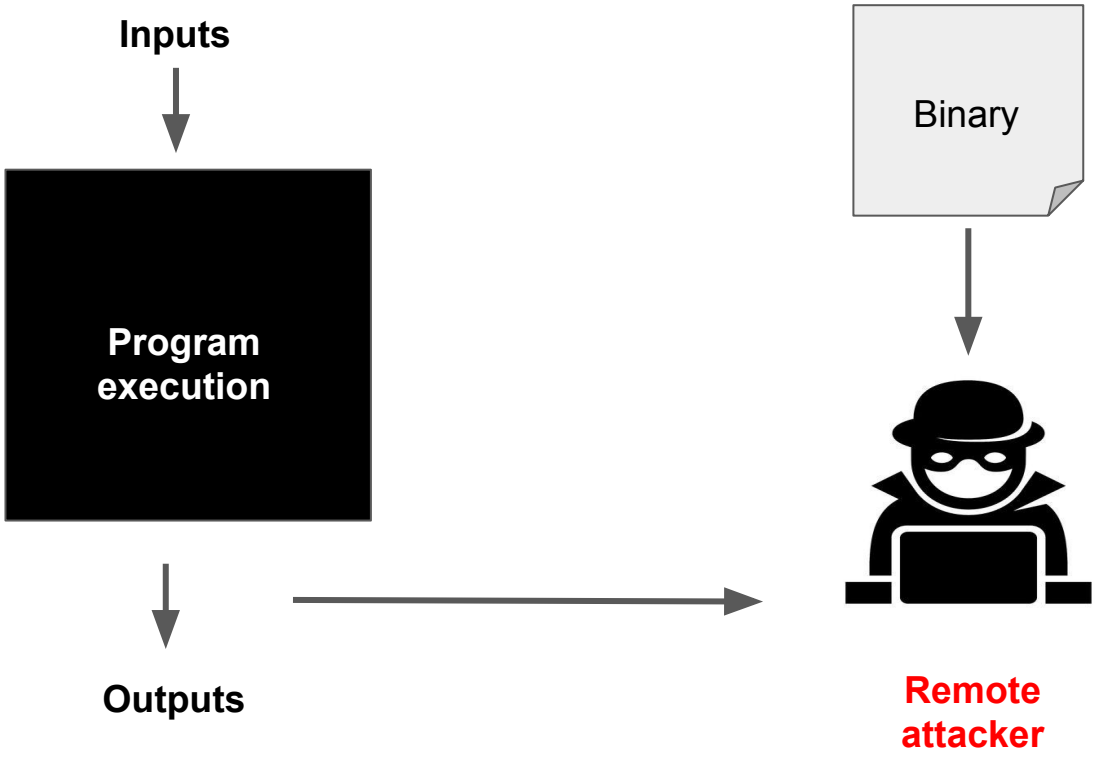
- We are interested in outputs leaking any transformation f_k of an address, i.e.,:

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

Assumptions

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

Attacker Model



Challenges

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

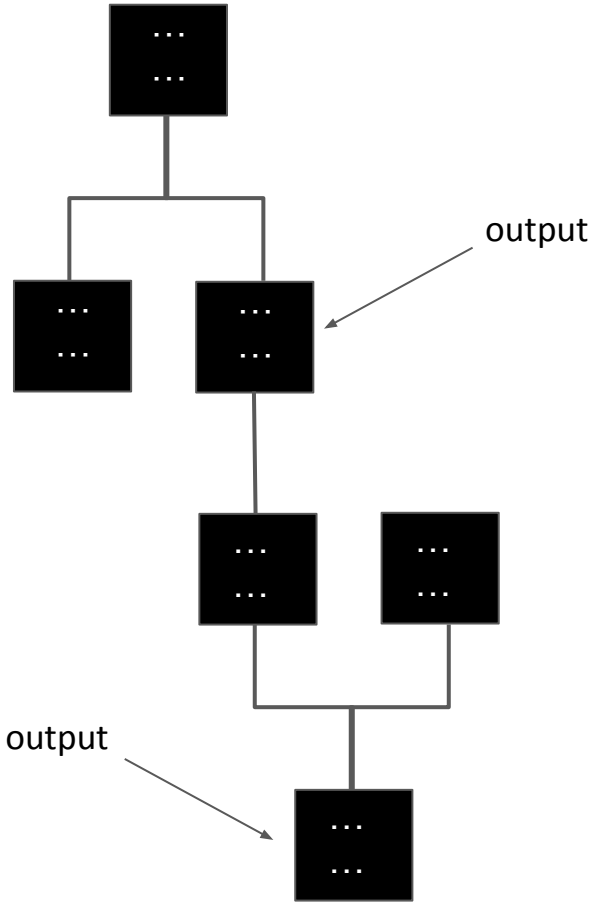
3. Leak identification

4. De-randomization

Static analysis: Path selection and address identification

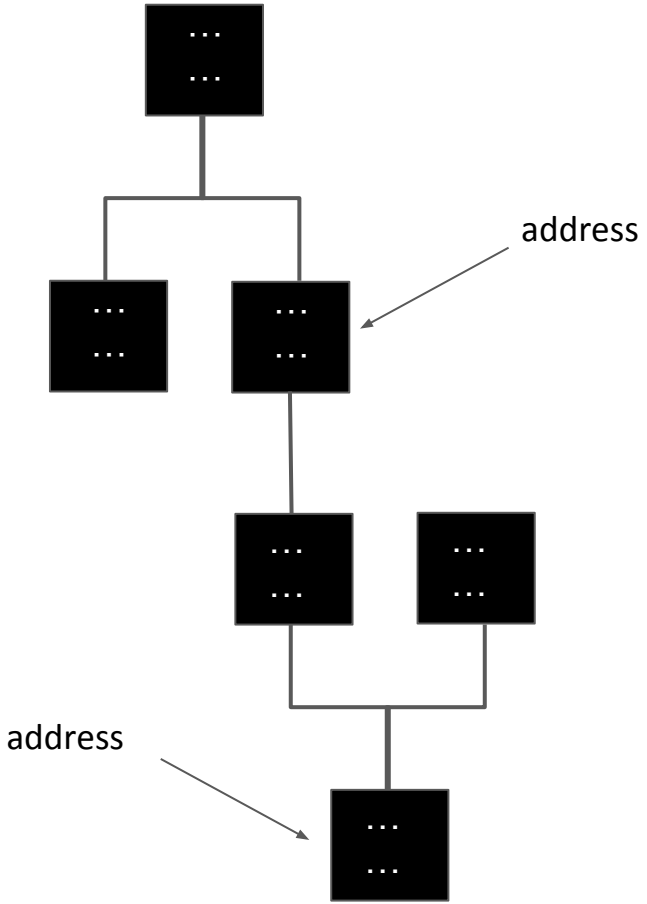
1: Output function identification

- 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

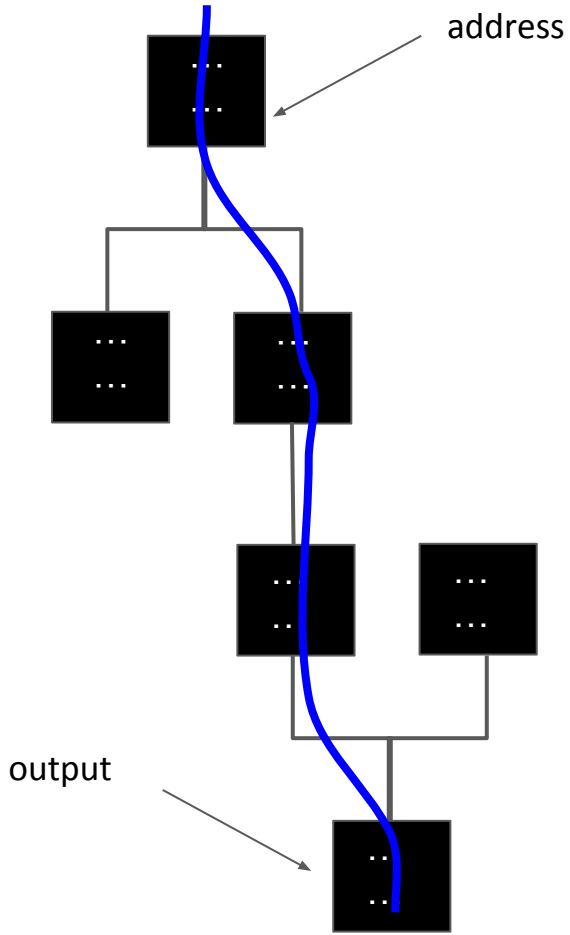
(1) Leverage known information (GOT, relocations, external function prototypes, e.g., return values of malloc() or mmap()).

(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?

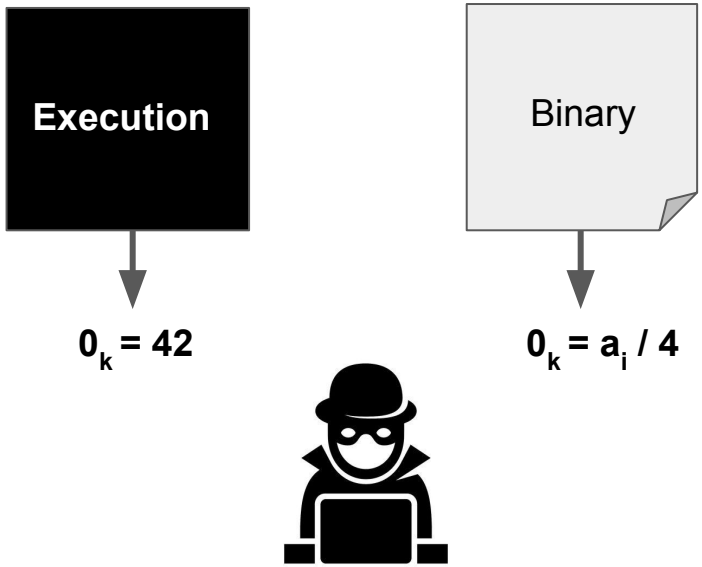
3: Leak Identification

- 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.




```
$ 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)

libXSLT (large library used by Firefox and Chrome)

OverlayFS (Linux Kernel)

- `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)

	CFG Nodes	Functions	Sinks	Leak Detected	Ground Truth
CTF binaries					
0x00ctf_17_left	72	1	3	✓	✓
a5afefd29d5dc067ed6507d78853c691	496	16	11	✓	x
defcon_16_heapfun4u	200	5	1	✓	✓
ez_pz	91	2	3	✓	✓
pwn1	318	1	1	✓	x
int3rrupted	327	6	4	✓	✓
libXSLT	76842	505	27	✓	✓
Overlayfs	1981	191	27	✓	✓

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.

?