The Security of
Address Space Layout Randomization
(ASLR)

Address Space Layout Randomization (ASLR)

- Traditional exploits require attacker to know addresses
  - Stack-based buffer overflows: location of shell code
  - return-to-libc: Library addresses
- Problem: the attacker knows the program layout on vulnerable host
  - Stack addresses
  - Heap addresses
  - Addresses of libraries
  - etc
- Solution: randomize the addresses of these items

Memory

Base address a

Executable
- Code
- Uninitialized data
- Initialized Data

Base address b

Mapped
- Heap
- Dynamic libraries
- Thread stacks
- Shared memory

Base address c

Stack
- Main stack

ASLR Randomization

a **+ 16 bit rand r1** b **+ 16 bit rand r2** c **+ 24 bit rand r3**

Executable	Mapped	Stack
• Code • Uninitialized data • Initialized Data	• Heap • Dynamic libraries • Thread stacks • Shared memory	• Main stack

Benefits

- ✓ Does not require recompiling programs
- ✓ Transparent to safe applications
- ✓ No/little overhead

When to randomize?

1. When a process starts
   – Constant randomization for all child processes

2. Periodically
   – After every fork, re-randomize child process

- Think about a web server

Security of ASLR

- Call an attempted attack with randomization guess ***x*** a probe
  - **x** is correct = Success = Root
  - Failure = detectable crash or no root
  - Assume 32-bit architecture, which works out to about 16 bits of randomness available for ASLR

- Scenario 1: A process is not randomized after each probe.

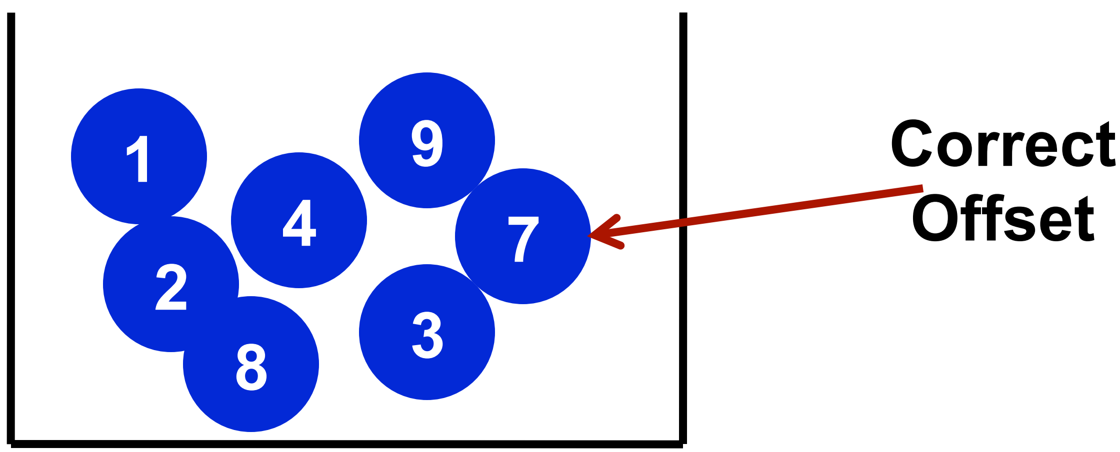
- Scenario 2: The address space is randomized after each probe.

What is the expected number of probes to hack the machine?

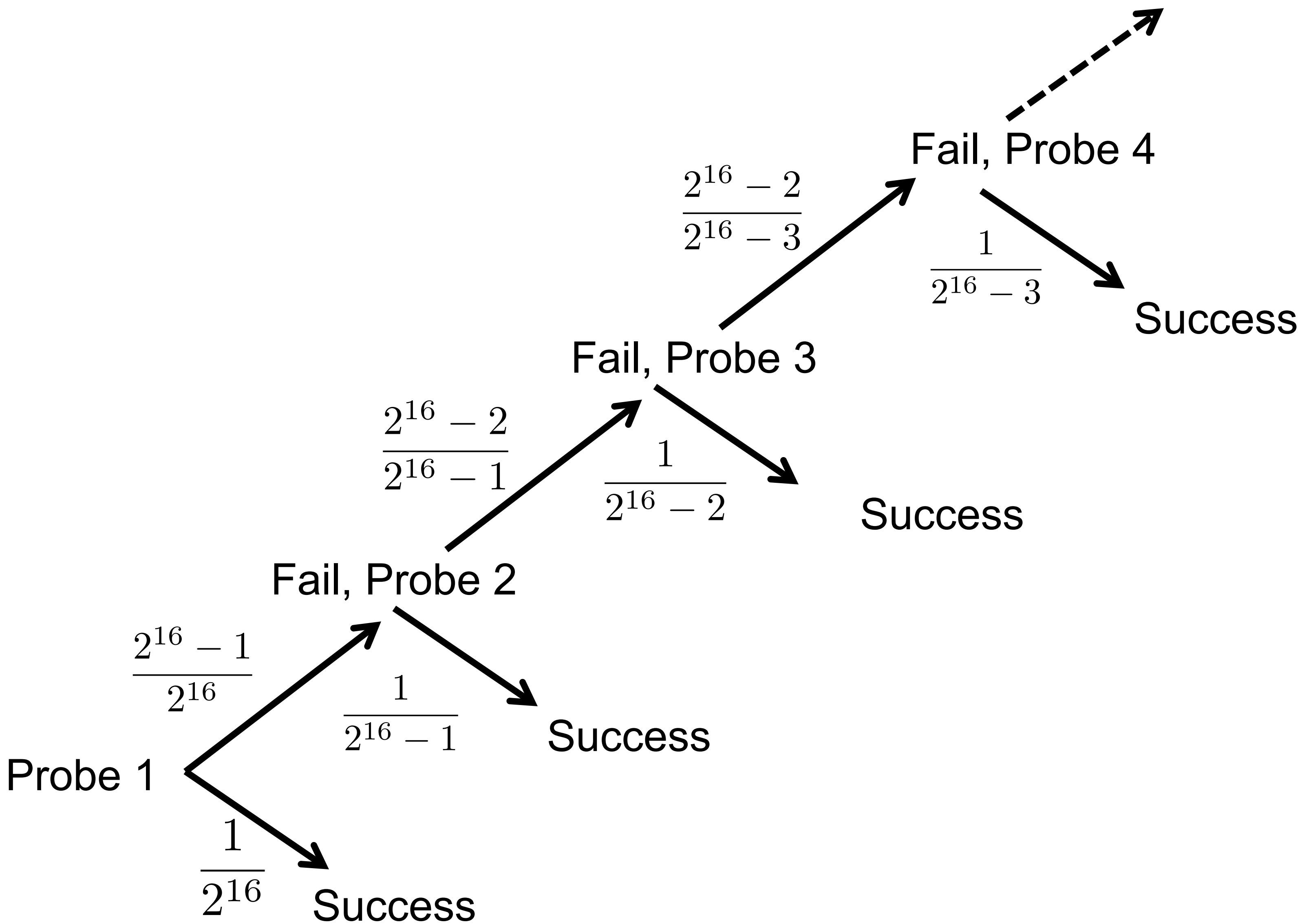
1. Pr[Success on exactly trial n]?
2. Pr[Success by trial n]?

Scenario 1: Not Randomized After Each Probe

- Pretend that each possible offset is written on a ball.
- There are 2^{16} balls
- This scenario is like selecting balls ***without replacement*** until we get the ball with the randomization offset written on it.



W/O Replacement:
Pr[Success on Exactly nth try]

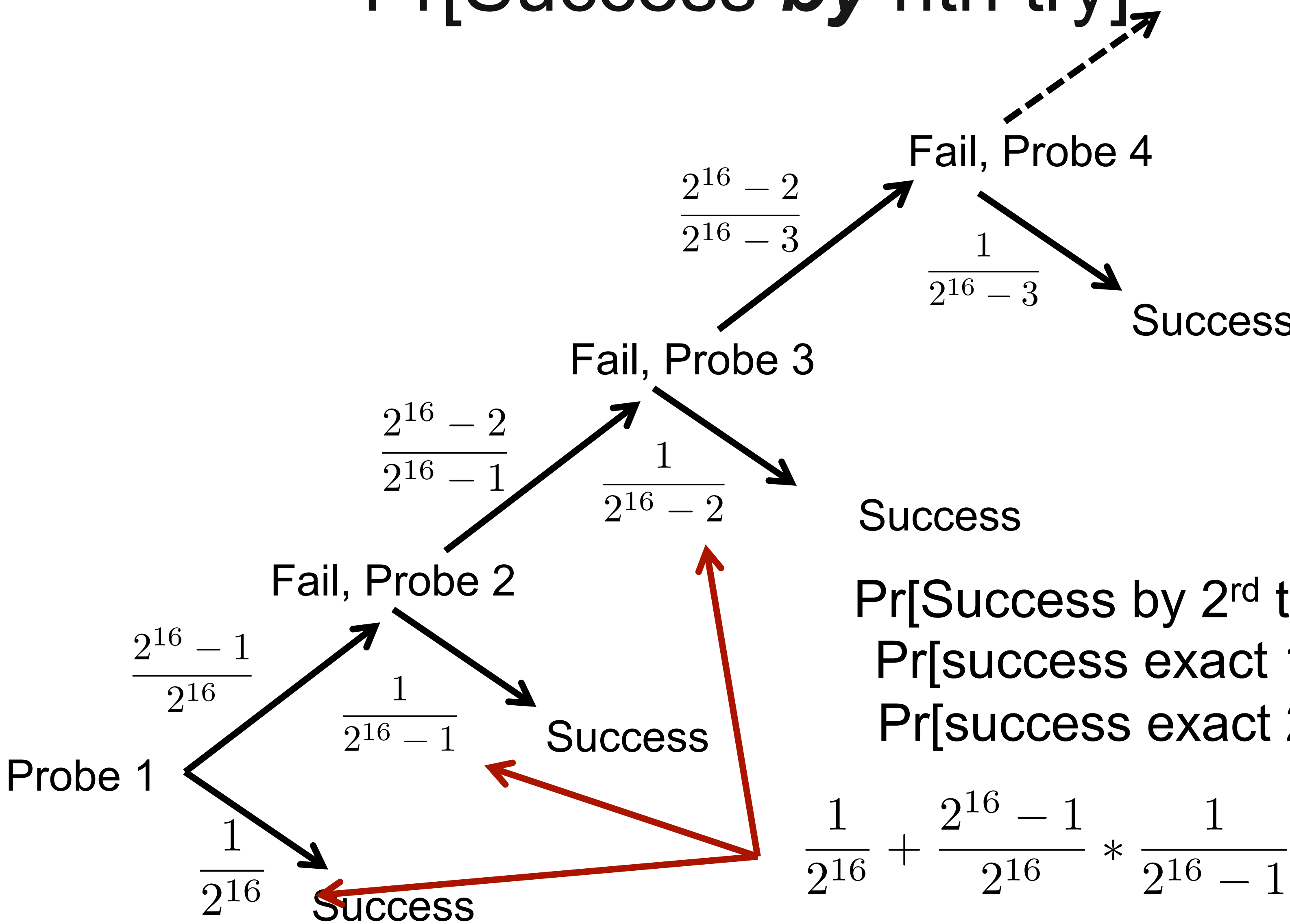


W/O Replacement:
Pr[Success on Exactly nth try]

$$\underbrace{\frac{2^{16}-1}{2^{16}} * \frac{2^{16}-2}{2^{16}-1} * \ldots * \frac{2^{16}-n-1}{2^{16}-n}}_{\text{Fail the first n-1 times}} * \frac{1}{2^{16}-n-1} = \frac{1}{2^{16}}$$

Succeed on nth trial

W/O Replacement:
Pr[Success ***by*** nth try]



Pr[Success by 2rd try] =
Pr[success exact 1st]+
Pr[success exact 2nd]

$$\frac{1}{2^{16}} + \frac{2^{16}-1}{2^{16}} * \frac{1}{2^{16}-1} = \frac{2}{2^{16}}$$

W/O Replacement:

Pr[Success ***by*** nth try] = $\dfrac{n}{2^{16}}$

What is the expected number of tries?

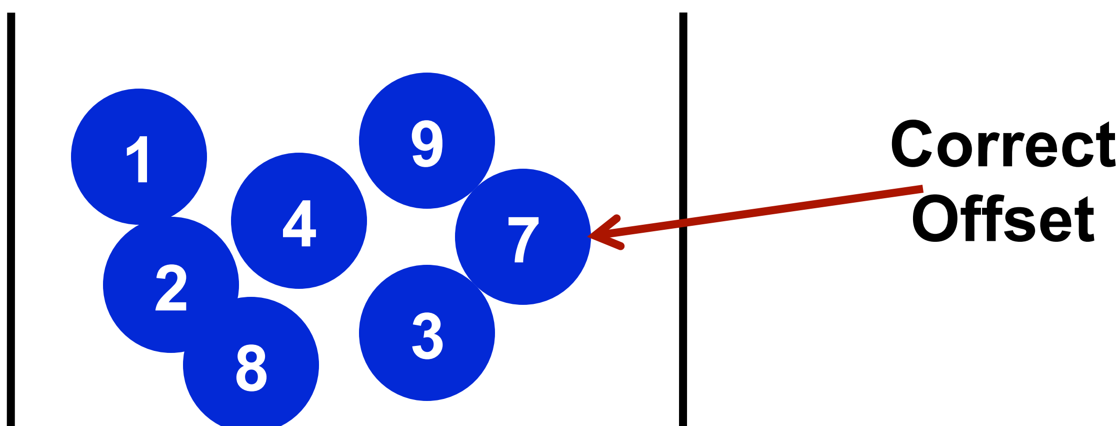
$$\text{Expectation} : \sum_{n=1}^{2^{16}} n * \frac{1}{2^{16}}$$

$$= \frac{1}{2^{16}} * \sum_{n=1}^{2^{16}} n$$

$$= \frac{2^{16}+1}{2} \approx 2^{n-1}$$

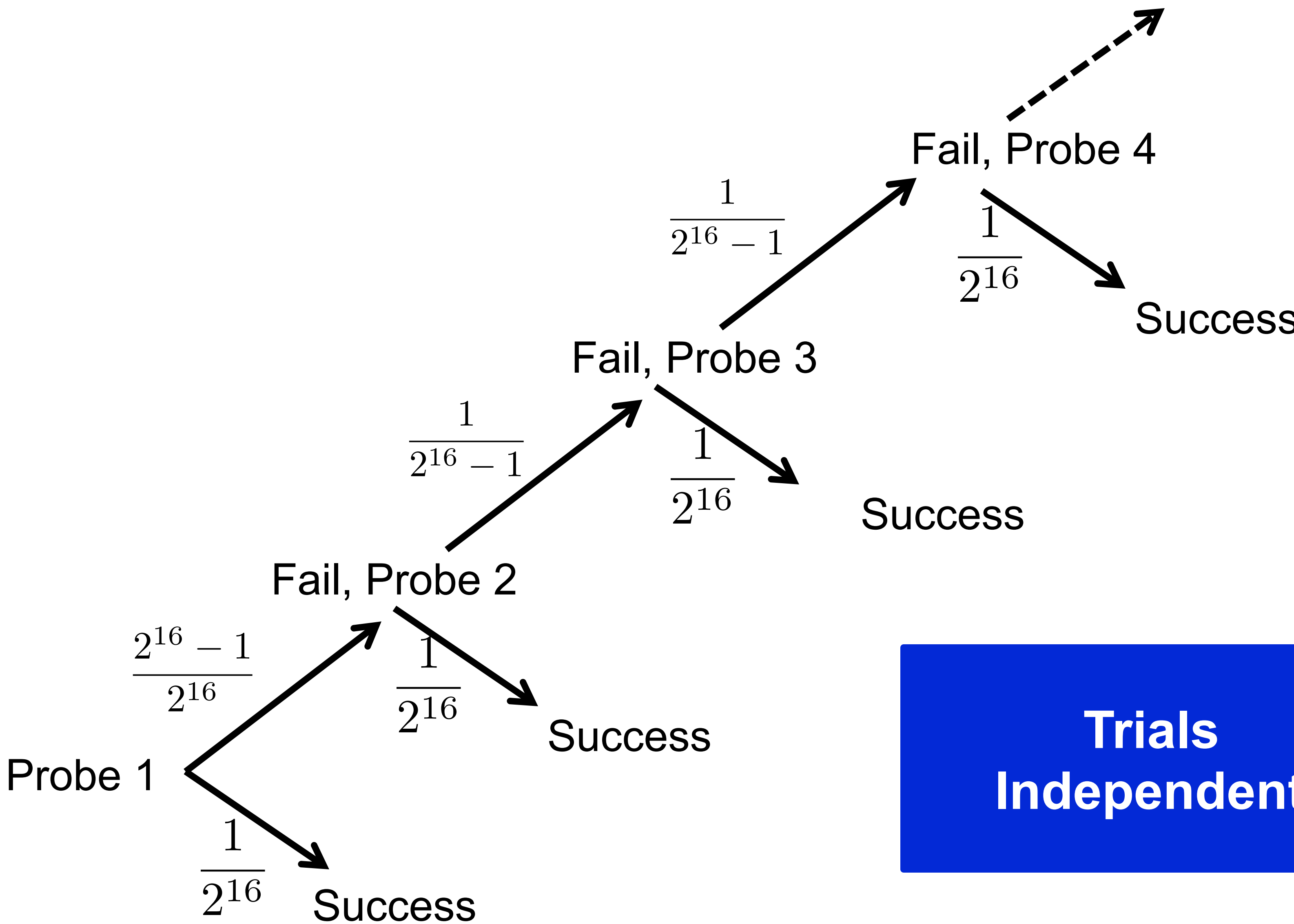
Scenario 2: Randomized After Each Probe

- Pretend that each possible offset is written on a ball.
- There are 2^{16} balls
- Re-randomizing is like selecting balls ***with replacement*** until we get the ball with the randomization offset written on it.



With Replacement

Pr[Success on exactly nth try]



Trials Independent!

With Replacement:

Pr[Success ***by*** nth try] = $\frac{1}{2^{16}}$

Expected number of probes = 2^{16}

Comparison

With Re-Randomization	Without Re-Randomization
Expected success in 2^{16} probes	Expected success in 2^{15} probes
For n bits of randomness: **2^n**	For n bits of randomness: **2^{n-1}**

Re-Randomization gives 1 bit of security

More Information

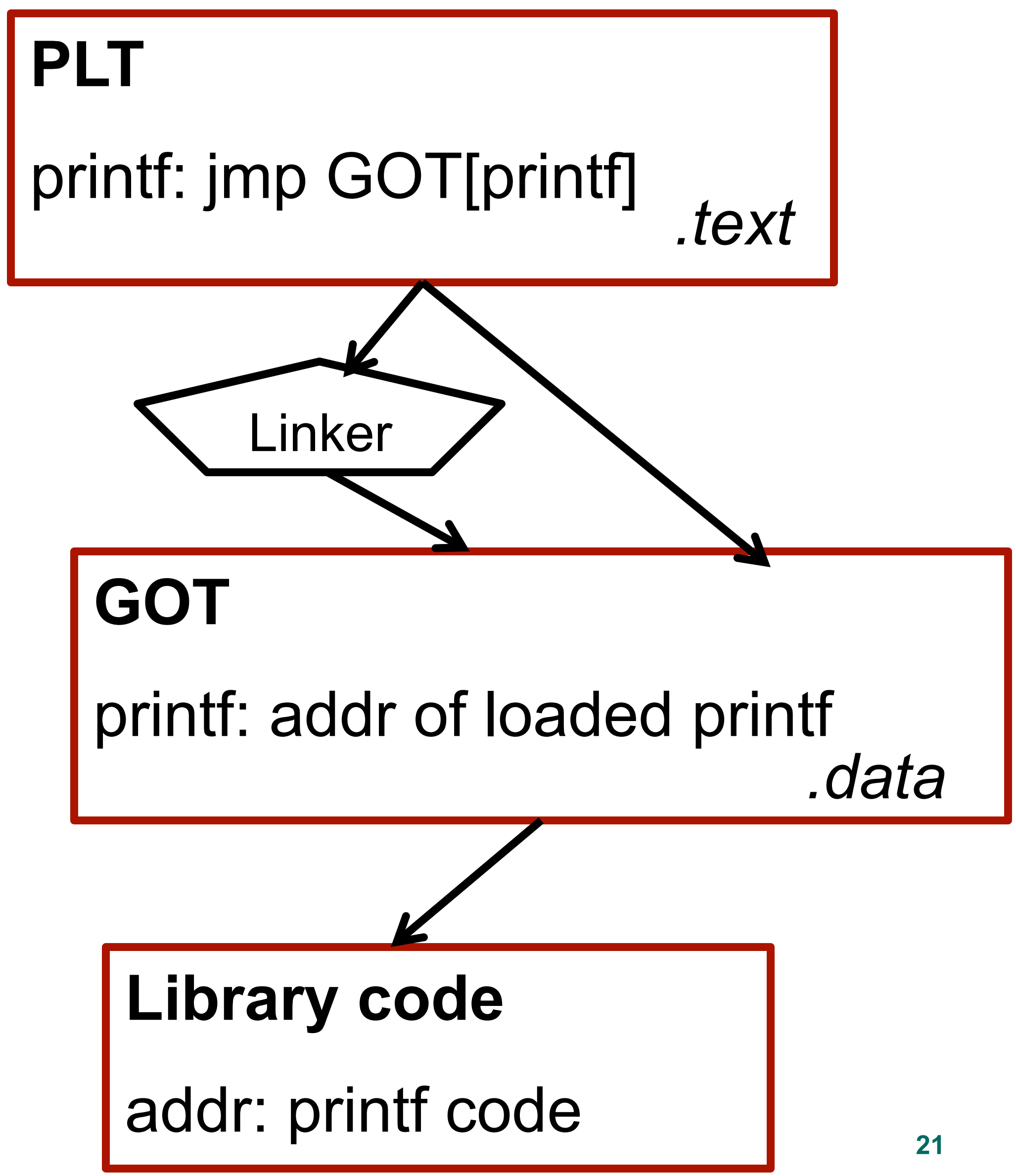
- “On the Effectiveness of Address-Space Randomization”
  - Shacham et al, at ACM CCS 2004
- “An Analysis of Address Space Layout Randomization on Windows Vista”
  - Ollie Whithouse, Symatec Research Whitepaper

Exploiting Non-Randomized Things

- Dynamically linked libraries are loaded at runtime. This is called *lazy binding*

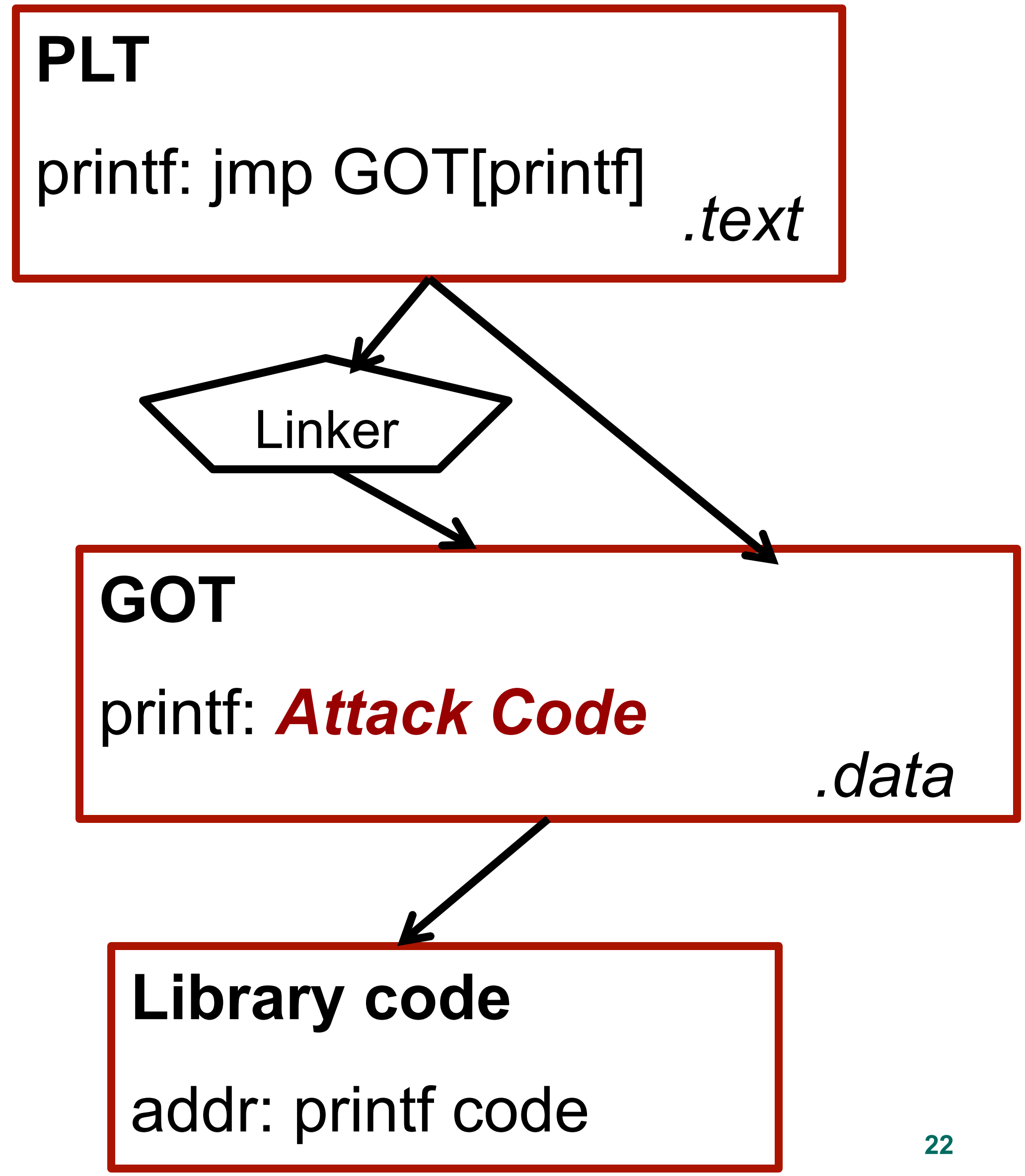
- Two important data structures
  - Global Offset Table (GOT)
  - Procedure Linkage Table (PLT)

```
…
printf("%s",input);
…
```



Exploiting Non-Randomized Things

```
strcpy(buf, input);
…
printf("%s",input);
…
```



Take Care